Fakultät II
Department für Informatik
Abteilung Eingebettete Hardware-/Software-Systeme

Application Mapping and Communication Synthesis for Object-Oriented Platform-Based Design

Dissertation
zur Erlangung des
Doktorgrades der Naturwissenschaften
(Doctor rerum naturalium)

vorgelegt von
Kim Grüttner
geboren am 06.03.1979 in Delmenhorst

Oldenburg 2015

Për Nikon dhe Idën time

Submitted: January 22, 2015

Advisor: Prof. Dr.-Ing. Wolfgang Nebel
First Reader: Prof. Dr. Achim Rettberg
Second Reader: Prof. Dr. Rainer Dömer

Defended: March 6, 2015

Abstract

Platform-based design of embedded systems on a chip consists of the parallel functional application specification, configuration of the hardware platform (i.e. connection of processing, memory and physical communication channels) and mapping of the application description on the processing, memory and communication resources of the hardware platform. The main contribution of this work is the seamless object-oriented modeling and automatic refinement of communication. In the application model, communication is specified by method calls on communication objects, independent from a physical communication channel. By means of mapping rules, these application model method calls are automatically transformed (synthesized) and implemented on the processing, memory and communication resources of the hardware platform. This approach enables the exploration and assessment of the impact of different platform configurations and mapping rules on functional and extra-functional properties.

Keywords: Embedded system on a chip, object-oriented communication, synthesis, remote method invocation, platform-based design

Kurzzusammenfassung

Der plattformbasierte Entwurf eingebetteter Ein-Chip-Systeme, besteht in der parallelen funktionalen Spezifikation der Applikation, Konfiguration der Hardwareplattform (d.h. Verbindung von Berechnungs- und Speicherelementen über physikalische Kommunikationskanäle), und der Abbildung der Applikationsbeschreibung auf die Berechnungs-, Speicher- und Kommunikationsressourcen der Hardwareplattform. Der Hauptbeitrag dieser Arbeit ist die durchgängige objektorientierte Modellierung und automatischen Verfeinerung der Kommunikation. Diese wird im Applikationsmodell mit Hilfe von Methodenaufrufen kommunizierender Objekte, unabhängig von einem physikalischen Übertragungskanal, modelliert. Mit Hilfe einer Abbildungsvorschrift werden diese Methodenaufrufe durch eine automatische Transformation (Synthese) auf Hardwareressourcen, Speicher und Kommunikationsleitungen der Hardwareplattform abgebildet. Dieses Verfahren ermöglicht eine Untersuchung und Bewertung unterschiedlicher Plattformkonfigurationen und Abbildungsvorschriften in Bezug auf funktionale und extra-funktionale Eigenschaften.

Schlagwörter: Eingebettetes Ein-Chip-System, objektorientierte Kommunikation, Synthese, Aufruf entfernter Methoden, plattformbasierter Entwurf

Contents

1 Introduction **13**
1.1 Embedded Systems on a Chip 14
1.2 Design Challenges 15
1.3 Contributions 17
1.4 Basic Idea 17
1.5 Outline 20

2 Goals of the Methodology **25**
2.1 Introduction 25
2.2 Embedded Systems on a Chip (SoC) 25
2.2.1 IP components 26
2.2.2 Software Processors 26
2.2.3 Dedicated Hardware 27
2.2.4 Memory 28
2.2.5 Communication Interfaces 28
2.2.6 Communication Networks 29
2.2.6.1 Point-To-Point Communication 29
2.2.6.2 Bus Communication 29
2.2.6.3 Shared Object Communication 30
2.2.7 Application mapping 31
2.3 Communication in Embedded System 31
2.3.1 Structural inter-component access techniques 32
2.3.1.1 Memory mapped I/O 32
2.3.1.2 Port I/O 33
2.3.2 Behavioral inter-component access techniques 33
2.3.2.1 Polling 33
2.3.2.2 Interrupts 33
2.3.2.3 DMA 34
2.4 Requirements on Communication-Centric Design 35
2.4.1 Modeling 36
2.4.2 Analysis 37
2.4.3 Synthesis 37
2.4.4 Implicit Requirements and Consequences 38
2.5 Summary 39

3 Terminology **41**
3.1 Introduction 41
3.2 Selected Mathematical Notations 41
3.3 Timed Automata 43
3.3.1 Definition 43
3.3.2 Graphical notation in Uppaal 44
3.3.3 Synchronous Value Passing 45
3.3.4 Properties 46
3.3.4.1 State Formulae 46
3.3.4.2 Reachability Properties 47
3.3.4.3 Safety Properties 47
3.3.4.4 Liveness Properties 47
3.4 Model of Computation, Architecture, Structure and Performance 48
3.4.1 Model of Computation (MoC) 48
3.4.2 Model of Architecture (MoA) 50
3.4.3 Model of Structure (MoS) 52
3.4.4 Model of Performance (MoP) 53
3.4.5 Summary 55
3.5 Methodology 57
3.5.1 Design flow 59
3.5.2 Simulation 59
3.5.3 Synthesis 60
3.5.4 Summary 60
3.6 System Level Design Representation 61
3.6.1 Language 61
3.6.2 Program State Machines 63
3.6.2.1 Program-States 63
3.6.2.2 Hierarchical composition 65
3.6.2.3 Communication 66
3.6.3 Sequential program representation 68

4 Related Work **73**
4.1 Introduction 73
4.2 Previous Work 75
4.2.1 Objective VHDL 75
4.2.2 Objective VHDL+ 76
4.2.3 SystemC-Plus 77
4.2.4 Discussion 78
4.3 Object-Oriented Communication Concepts in ESL Design 79
4.3.1 OOCL 79
4.3.2 CHSOM 80
4.3.3 Actor-oriented 80
4.3.4 CORBA- and Component-based 82
4.3.5 C++- and SystemC-based 84
4.3.6 Summary & Discussion 86
4.4 SoC communication modeling 87
4.4.1 SystemC TLM 88
4.4.2 GreenBus 88
4.4.3 Accuracy-Adaptive TLMs 88
4.4.4 OCP SystemC channels 89
4.4.5 STMicroelectronics TAC 89
4.4.6 $\text{SystemC}^{\text{SV}}$ 89
4.4.7 OCCN 89
4.4.8 IBM CoreConnect models 90
4.4.9 ARM AMBA models 90
4.4.10 CCATB AMBA models 90
4.4.11 ROM 90

4.4.12 NoC channels . . . 90
4.4.13 Summary & Discussion . . . 90
4.5 SoC Communication Synthesis . . . 91
4.5.1 SpecC-based . . . 92
4.5.2 SystemC-based . . . 93
4.5.3 Commercial SystemC and C/C++ synthesis tools . . . 93
4.5.3.1 SystemCrafter . . . 93
4.5.3.2 Handel-C . . . 94
4.5.3.3 C-to-Silicon . . . 94
4.5.3.4 Cynthesizer . . . 94
4.5.3.5 CatapultC . . . 95
4.5.3.6 Vivado . . . 95
4.5.3.7 eXCite . . . 96
4.5.4 Summary & Discussion . . . 96
4.6 Electronic System-Level Synthesis Methodologies . . . 97
4.6.1 Daedalus . . . 97
4.6.2 System-On-Chip Environment . . . 99
4.6.3 SystemCoDesigner . . . 100
4.6.4 Metropolis . . . 101
4.6.5 Koski . . . 102
4.6.6 PeaCE/HOPES . . . 102
4.6.7 Summary & Discussion . . . 103
4.7 Contribution of this work . . . 104

5 Methodology, Modeling Elements and Operational Semantics . . . 109
5.1 Introduction . . . 109
5.2 Abstraction Layers . . . 111
5.3 Object Model . . . 112
5.3.1 Basic Types . . . 113
5.3.2 Other types . . . 113
5.3.3 Array . . . 113
5.3.4 Class . . . 113
5.3.5 Interface class . . . 114
5.4 Behavioral Layer . . . 116
5.4.1 Introduction . . . 116
5.4.2 Modeling Elements . . . 118
5.4.2.1 Port . . . 118
5.4.2.2 Behavior . . . 119
5.4.2.3 Channels . . . 125
5.4.3 Operational Semantics . . . 132
5.4.3.1 Leaf Behavior . . . 132
5.4.3.2 Sequential composition (SEQ) . . . 134
5.4.3.3 Finite-state machine composition (FSM) . . . 135
5.4.3.4 Parallel composition (PAR) . . . 136
5.4.3.5 Pipeline composition (PIPE) . . . 138
5.4.3.6 Hierarchical composition . . . 142
5.4.3.7 Communication . . . 149
5.5 Application Layer . . . 154
5.5.1 Introduction . . . 154
5.5.2 Modeling Elements . . . 155
5.5.2.1 Actor . . . 156
5.5.2.2 Application Layer System . . . 157
5.5.2.3 Shared Objects . . . 158
5.5.3 Pre-defined Scheduling Algorithms . . . 162
5.5.3.1 Static Priority . . . 163
5.5.3.2 Ceiling Priority . . . 163
5.5.3.3 Round Robin . . . 163

5.5.3.4 Modified Round Robin . . . 164
5.5.4 Timing Annotations . . . 165
5.5.4.1 Shared Object annotations . . . 165
5.5.4.2 Actor annotation . . . 166
5.5.4.3 Timing estimation . . . 168
5.5.4.4 Timing analysis . . . 169
5.5.4.5 Properties of Timing Annotations . . . 171
5.5.4.6 Limitations . . . 172
5.5.5 Mapping rules . . . 173
5.5.6 Operational Semantics . . . 175
5.5.6.1 Actor . . . 176
5.5.6.2 Port . . . 178
5.5.6.3 Shared Object . . . 178
5.5.6.4 Putting it all together . . . 181
5.5.6.5 Properties . . . 184
5.6 Virtual Target Architecture Layer . . . 186
5.6.1 Introduction . . . 186
5.6.2 Modeling Elements . . . 186
5.6.2.1 Signal and Signal Port . . . 188
5.6.2.2 RMI Port . . . 189
5.6.2.3 Serializable Object . . . 189
5.6.2.4 Virtual Target Architecture Object . . . 190
5.6.2.5 Memory . . . 190
5.6.2.6 Software Socket . . . 195
5.6.2.7 Hardware Socket . . . 195
5.6.2.8 RMI Channel . . . 197
5.6.2.9 OSSS Channel . . . 203
5.6.2.10 IP Component . . . 211
5.6.2.11 Virtual System on Chip . . . 212
5.6.3 Mapping rules . . . 213
5.6.3.1 Add support for object serialization . . . 213
5.6.3.2 Software, Actor and Shared Object Behavioral-RT timing refinement . . . 214
5.6.3.3 RMI timing annotations . . . 214
5.6.3.4 Memory timing annotations . . . 216
5.6.4 Operational Semantics . . . 217
5.6.4.1 RMI Port . . . 218
5.6.4.2 Shared Object Socket . . . 220
5.6.4.3 OSSS-Channel . . . 220
5.6.4.4 Putting it all together . . . 223
5.6.4.5 Properties . . . 229
5.7 Target Platform . . . 230
5.7.1 Introduction . . . 230
5.7.2 Modeling Elements . . . 230
5.7.2.1 Software Processing . . . 231
5.7.2.2 Hardware Processing . . . 232
5.7.2.3 Memory . . . 232
5.7.2.4 Communication . . . 232
5.7.2.5 IP . . . 233
5.7.2.6 SoC . . . 233
5.8 Summary . . . 233

6 Simulation **235**
6.1 Introduction 235
6.2 Overview 237
6.2.1 SystemC^{TM} 237
6.2.2 OSSS 238
6.2.3 Behavioral Layer 239
6.2.4 Application Layer 239
6.2.4.1 Hardware/Software Intersection 239
6.2.4.2 Hardware Section 240
6.2.4.3 Software Section 240
6.2.5 Virtual Target Architecture Layer 240
6.3 Behavioural Layer 241
6.3.1 Introduction and motivation 241
6.3.2 Composite Behaviours 242
6.3.3 Communication 246
6.3.4 Hierarchical Behaviour composition 248
6.3.5 Current Limitations 250
6.4 Application Layer 251
6.4.1 Shared Object 251
6.4.1.1 Using Shared Objects 252
6.4.2 Adapter Socket 257
6.4.2.1 Using sockets 257
6.4.2.2 Restrictions 259
6.4.3 Software Task 260
6.4.3.1 Declaration of a Software Task 260
6.4.3.2 Instantiation of a Software Task 261
6.4.3.3 Using EETs for specifying the software timing behaviour 262
6.4.3.4 Using EETs and RETs for checking timing consistencies of Software Tasks 263
6.4.3.5 Restrictions when using Software Tasks 265
6.4.4 Hardware/Software Communication 266
6.4.5 Hardware Module 268
6.5 Virtual Target Architecture Layer 269
6.5.1 Architecture Class Library 269
6.5.2 Remote Method Invocation 270
6.5.2.1 The General Concept 270
6.5.2.2 RMI protocol stack 272
6.5.3 OSSS-Channels 282
6.5.3.1 Key Concepts 283
6.5.3.2 A Simple Point-To-Point Channel 284
6.5.3.3 A Channel with Arbitration 288
6.6 Mapping 291
6.6.1 Mapping the Consumer/Producer Design Example 292
6.6.1.1 The `osss_rmi_if<...>` interface stub 293
6.6.1.2 Serialisation of user-defined data types 295
6.6.1.3 The `osss_rmi_channel<...>` container for synthesisable OSSS-Channels 296
6.6.1.4 The `osss_object_socket<...>` container for Shared Objects 296
6.6.1.5 The final assembly phase 297
6.6.2 Architecture Exploration 300
6.7 Summary 303
6.7.1 Passive Modeling Elements 304
6.7.2 Active Modeling Elements 305
6.7.3 Mapping and Refinement 306
6.7.4 Review of Goals 307

7 Synthesis ... **311**
7.1 Introduction ... 311
7.2 Overall Flow ... 312
7.3 Parsing and Intermediate Representation ... 314
7.4 Target Platform Representation ... 315
7.4.1 Software Processor Block ... 316
7.4.2 Hardware Block ... 319
7.4.3 Memory ... 319
7.4.4 Communication Network ... 319
7.4.4.1 Point-To-Point Communication ... 321
7.4.4.2 Bus Communication ... 322
7.5 Platform Synthesis ... 322
7.5.1 Architectural Context Information ... 324
7.5.2 MHS and MSS Generation ... 327
7.5.3 UCF Generation ... 328
7.5.4 MPD and PAO Generation ... 328
7.5.5 OSSS ACI Generation ... 329
7.6 Software Synthesis ... 329
7.6.1 Introduction ... 329
7.6.2 The MicroBlaze Processor Subsystem ... 330
7.6.3 Supported Software Language Subset ... 331
7.6.4 The OSSS Software Library & RMI protocol stack ... 335
7.6.4.1 Application Layer ... 336
7.6.4.2 RMI Layer ... 339
7.6.4.3 Channel Layer ... 339
7.6.4.4 The native OPB Interface ... 339
7.6.4.5 The FSL Interface ... 341
7.6.5 Software Cross-Compilation ... 342
7.7 Custom Hardware Synthesis ... 343
7.7.1 *Fossy* ... 343
7.7.2 Synthesis Phases ... 344
7.7.2.1 Elaborator ... 344
7.7.2.2 Channel Synthesis ... 345
7.7.2.3 Shared Object Synthesis ... 345
7.7.2.4 Class Synthesis ... 346
7.7.2.5 Integer Type Synthesis ... 347
7.7.2.6 Delaborator ... 350
7.7.2.7 Code Generator ... 350
7.8 Shared Object Hardware Synthesis ... 350
7.8.1 Overview ... 350
7.8.2 RMI Controller ... 352
7.8.3 Interface Blocks ... 353
7.8.4 Scheduler ... 354
7.8.5 Guard Evaluator ... 354
7.8.6 Behavior Process ... 354
7.8.7 Potential Extensions and Optimizations ... 354
7.8.8 Hardware Client ... 355
7.9 Back-End Synthesis ... 360
7.9.1 Integration into Xilinx Flow ... 362
7.9.1.1 Integrated Software Environment (ISE) ... 362
7.9.1.2 Xilinx Platform Studio (XPS) ... 363
7.10 Summary ... 364

8 Experiments . . . 369
8.1 Introduction . . . 369
8.2 JPEG Encoder . . . 369
8.2.1 Goals of this experiment . . . 369
8.2.2 Introduction to JPEG . . . 369
8.2.3 JPEG encoder model . . . 372
8.2.4 Results . . . 374
8.2.5 Conclusion . . . 375
8.3 Adaptive Video Filter . . . 377
8.3.1 Goals of this experiment . . . 377
8.3.2 Model Composition . . . 377
8.3.3 Modeling in OSSS . . . 378
8.3.3.1 Behavior Layer Model . . . 378
8.3.3.2 Application Layer Model . . . 381
8.3.3.3 Virtual Target Architecture Model . . . 383
8.3.3.4 Target Platform Layer . . . 383
8.3.4 Conclusion . . . 383
8.4 NightView Video Filter . . . 383
8.4.1 Goals of this experiment . . . 383
8.4.2 Introduction & Motivation . . . 384
8.4.3 The NightView Application . . . 385
8.4.4 Target Platform . . . 386
8.4.5 Video processing algorithms . . . 386
8.4.5.1 Sobel filter . . . 386
8.4.5.2 Gamma correction filter . . . 387
8.4.5.3 Filter configuration examples . . . 387
8.4.6 Design flow . . . 387
8.4.7 Modeling in OSSS . . . 387
8.4.7.1 Application Layer Model (N2b) . . . 388
8.4.7.2 Virtual Target Architecture Layer Model (N3b) . . . 392
8.4.8 Evaluation . . . 394
8.4.8.1 Simulation performance . . . 395
8.4.8.2 Model complexity . . . 395
8.4.8.3 Chip area . . . 396
8.4.9 Conclusion . . . 396
8.5 MP3 Decoder . . . 397
8.5.1 Goals of this experiment . . . 397
8.5.2 Introduction to MP3 decoding . . . 397
8.5.3 Modeling in OSSS . . . 398
8.5.3.1 Profiling . . . 399
8.5.3.2 Application Layer Model . . . 400
8.5.3.3 Virtual Target Architecture Layer Model . . . 401
8.5.3.4 Implementation Model . . . 402
8.5.4 Results . . . 402
8.5.4.1 Software implementation . . . 403
8.5.4.2 Hardware/Software implementation . . . 404
8.5.4.3 RMI overhead . . . 405
8.5.5 Conclusion . . . 406
8.6 IPv4 Packet Forwarding Switch . . . 406
8.6.1 Goals of this experiment . . . 406
8.6.2 Introduction & Motivation . . . 407
8.6.3 Modeling in OSSS . . . 407
8.6.4 Synthesis . . . 409
8.6.5 Conclusion . . . 417
8.7 JPEG 2000 Decoder . . . 419
8.7.1 Goals of this experiment . . . 419
8.7.2 Introduction . . . 419

8.7.3 Modeling in OSSS . . . 420
8.7.3.1 Application Layer Model . . . 420
8.7.3.2 Architecture Layer Model . . . 422
8.7.4 Implementation Models . . . 424
8.7.4.1 IDWT Reference Models . . . 426
8.7.4.2 *Fossy* Generated Models . . . 426
8.7.4.3 Comparison . . . 426
8.7.5 Conclusion . . . 427
8.8 Summary . . . 427

9 Conclusion . . . 431
9.1 Review of Goals . . . 434
9.2 Limitations and Future Work . . . 438

A Survey . . . 445

B Timed Automata Templates and Examples . . . 461
B.1 Used scheduling algorithms . . . 461
B.2 Application Layer TA example . . . 463
B.3 Virtual Target Architecture Layer TA example . . . 465

C Pre-defined Shared Objects . . . 469

D I^{2}C Protocol OSSS Channel Implementation . . . 475
D.1 Introduction . . . 475
D.2 The I^{2}C Bus Protocol . . . 476
D.2.1 Data Transfer from Master to Slave . . . 476
D.2.2 Data Transfer from Slave to Master . . . 476
D.3 OSSS Channel implementation . . . 478

E Supported Target Platforms . . . 483
E.1 Supported FPGAs . . . 483
E.1.1 Virtex-4 . . . 483
E.1.2 Virtex-II Pro . . . 484
E.2 Supported Prototyping and Development Boards . . . 485
E.2.1 The Xilinx ML401 Evaluation Platform . . . 485
E.2.2 The Xilinx University Program Virtex-II Pro Development Board . . . 488
E.3 Basic IP components . . . 490
E.3.1 MicroBlaze Local Memory . . . 490
E.3.2 Interrupt Controller . . . 491
E.3.3 Timer . . . 492
E.3.4 Universal Asynchronous Receiver Transmitter (UART) . . . 492
E.3.5 Microprocessor Debug Module (MDM) . . . 493
E.3.6 On-Chip Peripheral Bus (OPB) . . . 493
E.3.6.1 Features . . . 494
E.3.6.2 FPGA implementation supported features . . . 494
E.3.6.3 Connection . . . 495
E.3.6.4 Arbitration . . . 495
E.3.7 Intellectual Property Interface (IPIF) . . . 496

F Synthesis Subset . . . 499
F.1 Compatibility to the SystemC Synthesisable Subset . . . 499
F.2 Coding Guidelines . . . 501
F.2.1 Design Hierarchy . . . 501
F.2.1.1 Modules . . . 501
F.2.1.2 Constructors . . . 502
F.2.1.3 Ports . . . 503
F.2.1.4 Signals and Channels . . . 503
F.2.1.5 Bindings . . . 504
F.2.2 Processes . . . 504
F.2.2.1 Effect of `wait()` usage on the number of states . . . 506
F.2.2.2 Reset . . . 507
F.2.3 Datatypes . . . 508
F.2.4 Statements and Expressions . . . 509
F.2.5 Classes and Inheritance . . . 509
F.2.6 Templates . . . 510
F.2.7 Namespaces . . . 510
F.2.8 Polymorphic Objects . . . 510
F.2.9 Shared Objects . . . 510
F.2.10 Non-Synthesisable . . . 511

G Integrated Development Environment . . . 513
G.1 Introduction . . . 513
G.2 Using the Eclipse CDT with FOSSY integration . . . 514
G.2.1 Project Navigator . . . 514
G.2.2 The SystemC Alarm Clock Example . . . 515
G.2.3 Building the Pre-Synthesis Model . . . 516
G.2.4 Building the Post-Synthesis Model . . . 519
G.2.5 Generating VHDL code . . . 520
G.3 Creating a custom project . . . 520

H OSSS Behavior Graphs . . . 521

Bibliography . . . 525

Curriculum Vitæ . . . 539

CHAPTER 1

Introduction

This work describes the whole Oldenburg System Synthesis Subset (OSSS) modeling language and methodology (see Chapter 5), simulator (see Chapter 6) and system synthesis (see Chapter 7). OSSS has been developed at OFFIS – Institute for Information Technology, during a series of European research projects:

ODETTE: **O**bject-oriented co-**DE**sign and functional **T**est **TE**chniques [224]

ICODES: **I**nterface- and **CO**mmunication-based **D**esign of **E**mbedded **S**ystems [223]

ANDRES: **AN**alysis and **D**esign of run-time **RE**configurable, heterogeneous **S**ystems [220]

In a national funded project at the Carl von Ossietzky University, called PolyDyn (**Poly**morphic Objects for **Dyn**amically reconfigurable FPGAs) [226], further extensions for modeling polymorphism in hardware and its implementation using dynamically reconfigurable Field Programmable Gate Arrays (FPGAs) have been performed. After completion of the ANDRES project, parts of the OSSS simulation and synthesis technology have been commercialized by CoSynth GmbH & Co. KG [216].

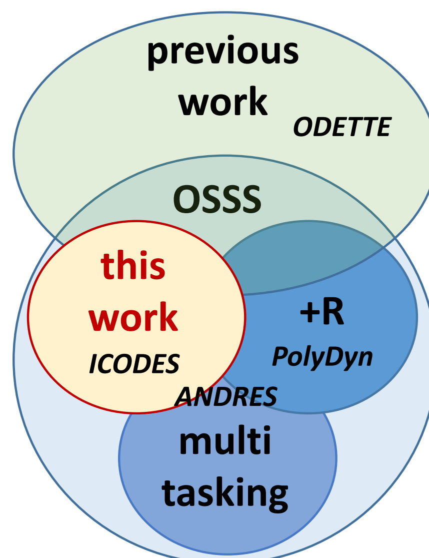


Figure 1.1: Overview of related projects and classification of this work

In particular, this work covers the part of OSSS developed during ICODES and ANDRES, without covering the polymorphic extensions developed in PolyDyn and ANDRES, as shown in Figure 1.1. More details about previous and related work can be found in Chapter 4.

The contribution of this work is the definition of the multi-layer OSSS modeling methodology, supported refinement, simulation and synthesis of object-oriented communication through Remote Method Invocation (RMI) on top of target System on a Chip specific processing and communication elements.

Before further defining the aims of this work and its main contributions in Section 1.3, a brief introduction of the considered class of embedded systems and its associated design challenges relevant for this thesis is given in Section 1.1 and Section 1.2. After defining the aims of this thesis in Section 1.3, the basic idea for achieving these aims is described in Section 1.4. This introduction closes with an outline of this thesis, describing its overall organization and structure in Section 1.5.

1.1 Embedded Systems on a Chip

Embedded systems in general represent a set of information processing hardware/software systems, which are integrated in mostly complex technical systems to perform central control functions. The attribute "embedded" reflects the fact that embedded systems are highly integrated into complex technical systems and are mostly hidden from the human users of these systems. The attribute "on a Chip" reflects that the system is fully implemented on a single System on a Chip.

Today, highly integrated embedded Systems on a Chip (SoC) have a wide range of usage in our daily life, such as telecommunications and automotive. These modern embedded systems consist of several IP[1] blocks plus a few custom components and often use pre-defined technology platforms to implement them. Existing and upcoming technologies offer ever more possibilities to cope with the increasing application requirements. Nevertheless, efficiently designing such systems remains a major challenge since electronic design automation tools and methodologies cannot keep pace.

In our definition, embedded systems consist of an arbitrary number of the following components:

- custom software components to be executed on software processors,
- custom hardware components that may perform computationally intensive (e.g. high-speed data processing) or time critical computations (e.g. custom real-time communication protocols),
- pre-designed hardware components, like software processing elements (e.g. ARM, PowerPC), hardware processing elements (e.g. AES, DCT, FFT), memory controllers, memories (e.g. SRAM, DRAM, Flash), communication controllers (e.g. USB, Ethernet, I²C), and on-chip communication fabrics (e.g. bus, cross-bar, FIFOs) and
- pre-designed software components, like data processing and control algorithms, operating systems and device drivers.

It has been observed that the integration of these different pre-designed components into an efficiently working system is much more difficult than the design of a single component. This has massively influenced the principle of platform-based design [71].

In practice, most embedded systems are implemented on so-called technology platforms. These platforms provide a basic configuration consisting of processors, memories, special hardware resources (or accelerators), communication peripherals, and communication resources (like buses, special cross-bar switches or Network-on-Chips) to connect them all together. The application is implemented on top of this platform configuration. Some platforms provide advanced configuration and parametrization and offer flexibility for the implementation of different applications (e.g. platform FPGAs [232]).

[1] Intellectual Property

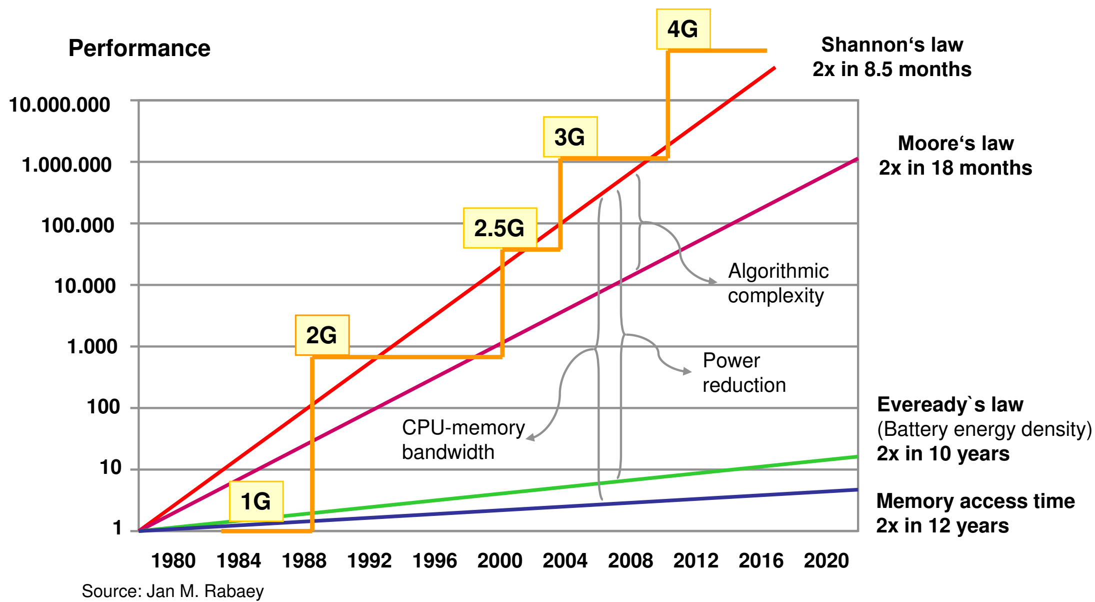


Figure 1.2: Key Technology Gaps

1.2 Design Challenges

From a technical point of view the design of embedded HW/SW systems involves all steps from the initial specification of the application - *what is the system supposed to do* - to the efficient implementation on a technology platform - *how is it implemented*. The challenge consists in finding methods for the description and transformation/refinement of the initial specification to an efficient implementation.

From the industrial point of view, the requirements for the design of embedded systems are very tight. Targeted development costs and time-to-market needs to be matched, but at the same time functional and extra-functional product requirements need to be fulfilled. Consequences for the design process are:

- modeling on the highest possible level of abstraction (mastering of complexity),
- most accurate analysis of system characteristics (prevention of costly re-design cycles),
- high degree of automation (raising efficiency and prevention of errors) and
- re-use of pre-existing hardware and software components.

Figure 1.2 shows the key challenges for the embedded mobile device industry. The performance growth of mobile phones per generation from analogue (1G) over GSM (2G) to high bandwidth UMTS (3G) and LTE (4G) follows Shannon's Law [205, 204], predicting an application complexity doubling every 8.5 months. For more than 30 years now the integration density of integrated circuits approximately doubles every 18 months (Moore's Law [203]). Network connection speeds doubles every 21 months (Nielsen's Law [171], not shown in Figure 1.2). As a corollary, Nielsen noted that, since the network connection speed rate is growing slower than that predicted by Moore's Law, user experience would remain communication bandwidth-bound. Battery makers need 5 to 10 years to achieve comparable increase in power density, and memory access time performance doubles every 12 years only. The gaps in this figure define the challenges, which the Electronic Design Automation (EDA) industry is facing:

1. Algorithmic complexity gap,
2. Microprocessor and memory bandwidth gap, and
3. Power reduction gap.

In order to keep up with the rapid technological advances, system design methodologies and EDA support were always forced to evolve in the past. Without the support of design methodologies and appropriate tool support, the design gaps are becoming larger:

- We observe a rising complexity of applications and execution platforms. The gap between these complexities boosts the uncertainty of platform selection and application to platform mapping. It requires a platform independent application description and EDA tools and methods able to support application to target architecture mapping and synthesis. For efficient targeting and exploring different mappings and execution platforms, guided or even fully automatic synthesis is required. The simulation of the mapping result (application running on the targeted platform) needs to be fast enough for running it in a design-space exploration loop to analyze at least the most meaningful parameters within the order of hours.

- Since the power and especially temperature is the next main limiting factor [16], a balance between performance and power needs to be found early in the design process. This can only be performed under explicit consideration of the interaction of application and target platform. To ensure a sufficient power and temperature control, EDA methods and tools need to be able to represent the extra-functional properties timing, power and temperature on all different levels of abstraction and in all executable models.

- To cope with the memory access time gap, smarter application specific memory organization is required. Treating memory as a homogeneous monolithic block is not possible anymore. For high-performance and power efficient systems, the parallelization and possible distributed memory schemes, including memory access scheduling, need to be supported on all stages of the design process. EDA methods and tools shall be able to model computation and state/storage explicitly. Mapping of variables and objects of an application to explicit memory elements of the target architecture needs to be supported.

In custom hardware design, advances in performance and power consumption have been mainly influenced by new technologies and an evolution in design methodology. The latter was achieved by several important steps in climbing up the level of abstraction for the design entry. All these steps gave the productivity in hardware design a great boost and made it possible to manage the steadily growing complexity of integrated circuits. Software processing units like micro-controllers, SIMD processors or DSPs have been made more efficient. Modern platforms support advanced power management capabilities with dynamic frequency scaling (sometimes independent for processing and memory subsystem) and power islands that needs to be effectively controlled to maximize the optimization potential.

Thus, there is a need for a new evolving design entry at system level where hardware and software are described in the same way. The latest trends in software engineering and hardware design have some commonalities, which might be not apparent on the first look. In the hardware and software world we observe the introduction of methodologies that separate the functionality/algorithm from the concrete implementation platform. The hardware world calls this evolution ESL; the software world calls this model-based design or MDE (Model-Driven Engineering), with Model-Driven Architecture (MDA). Both follow the Y-chart approach (separation of functionality - what - from the implementation platform - how).

The availability of this new design entry in conjunction with the traditional bottom-up approach defines a new viewpoint for the design of embedded systems: How to find a mapping and implementation of an application onto an execution platform that fulfills all functional and extra-functional requirements at minimal cost. To avoid expensive redesigns and costly code modifications, the platform decision should be done before investing money into a concrete target platform. For this purpose reliable information about the execution behavior of the application running on the platform in terms of functionality, performance and power consumption are mandatory.

1.3 Contributions

The main goal of this thesis is to provide an efficient methodology for mapping object-oriented applications to embedded System on Chip architectures, including automatic communication interface refinement and synthesis. The main distinguishing feature of this thesis is the seamless support of application-level method call communication on all levels of abstraction, down to a "silicon-proven" implementation on a FPGA platform.

The main contributions are:

1. Definition of a **multi-layer executable parallel object-oriented application description** that supports **custom application-level method call communication**.
2. Definition of an **object-oriented model of target platform** that represents processing, communication and memory resources.
3. Definition of a relation to express computation, communication and memory **mappings of the application to a "bare metal[2]" target platform, retaining application-level method calls** through Remote Method Invocation (RMI) techniques.
4. Definition of operational semantics and implementation of a pre- and post-target platform mapping **simulation for checking functionality and timing**.
5. Definition of timed automata formal model of pure application and target platform mapped application model to enable **analysis of safety** (deadlock, method call duration, mutual exclusiveness of method calls, buffer size limitations) **and liveliness properties** (guarantee that a method call will be served).
6. Definition of synthesis semantics and **proof-of-concept synthesis of mapped application on FPGA target platform** (enabling memory space and hardware area analysis) supporting RMI synthesis for hardware/hardware and hardware/software communication.

1.4 Basic Idea

The section sketches the basic idea for achieving the aims described above. Given a parallel object-oriented description of an embedded system application with functional, timing and performance requirements, an implementation on a hardware platform should be realized. The implementation on the hardware platform shall fulfill the functional and timing requirements of the application description (see Figure 1.3).

In this work, the *parallel object-oriented application description* consists of Actors and Objects, based on the active object design pattern, that decouples method execution from method invocation for objects that reside in their own thread of control (called Actors)[163]. This way concurrency is introduced and concurrent method invocation requests of different Actors on the same Object are handled by a scheduler. This application is target platform independent and can be executed (i.e. simulated) for analysis using a discrete event simulator.

The used *"bare metal" target platform model* consists of processing, memory and communication elements. These elements can either be generic or existing pre-designed or custom designed components. The target platform models are represented as classes and organized in an *architecture class library*. Different target platforms can be assembled by instantiating and connecting elements from the architecture class library. The platform model itself is non-executable but offers communication and memory services for a mapped application model:

Remote Method Invocation (RMI) Service enables to call a method of a remote object, i.e. an object that is executed on another processing element than the calling object. The RMI service handles the whole communication protocol over different platform communication elements and supports *serialization* and *de-serialization* (sometimes also called *marshalling* and *unmarshalling*) of a method's arguments and return values.

[2] *Bare metal* (or *bare machine*), in computer terminology, means a computer without an operating system.

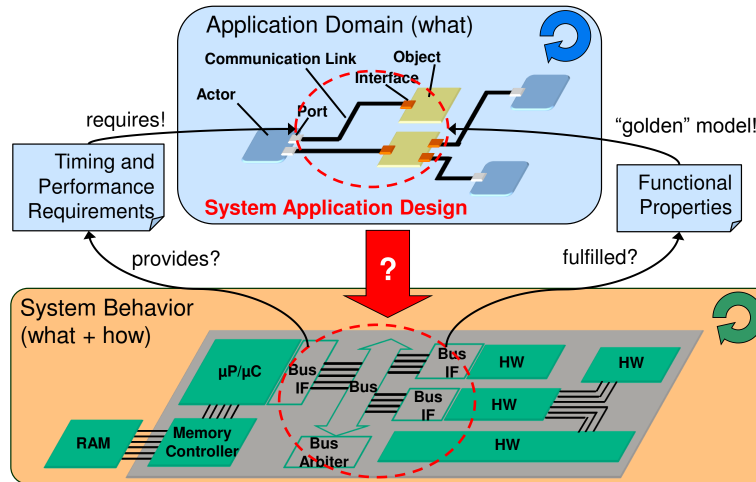


Figure 1.3: Problem Statement

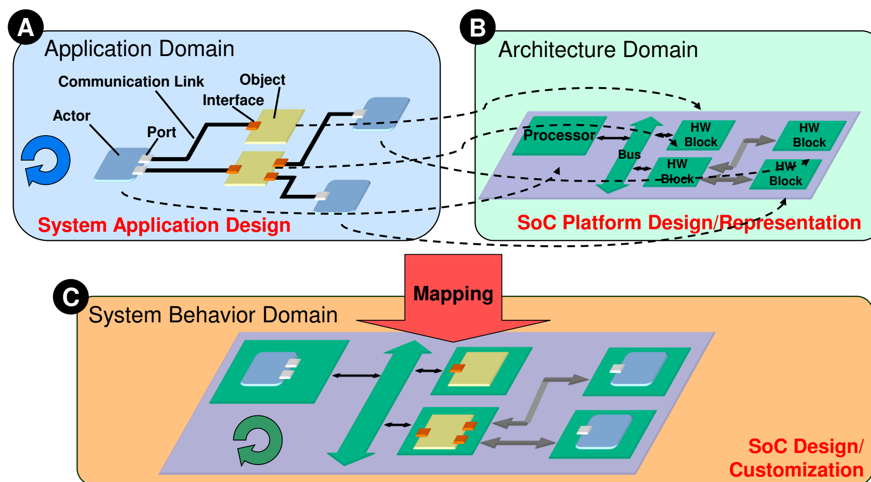


Figure 1.4: Basic Idea: Virtual Platform-Based Design

Memory Service enables user-defined access to user-defined data types (incl. objects) mapped into linear memory elements of the target platform.

To combine and evaluate the execution of the application model on the target platform, we are using *Platform-Based Design (PBD)* techniques (see Section 3.5 for more details on PBD). Figure 1.4 depicts the combination of application, target platform with the goal to obtain an executable virtual platform model of the application.

The PBD model enables a separate description of the application Ⓐ, the target platform Ⓑ and the mapping between application model and target platform model elements, resulting in an executable system prototype Ⓒ. The executable system prototype represents the execution times of the Actors executing on the respective processing elements, the communication times for accessing remote objects. The communication times include access arbitration delay, parameter serialization and de-serialization times, parameter transfer delay over platform communication elements (shared bus or dedicated point-to-point channel), and the method execution time itself. The RMI service is part of the target platform communication elements.

Figure 1.5 gives an introduction to the basic idea of the RMI-based object-oriented communication mapping. In the Application Model, communication and synchronization is preformed by user-defined method calls. In the example of Figure 1.5 a) the `EngineControl` provides the public method `accelerate()`. This service is provided by the aggregate `PowerBand`. In this simplified example, the `EngineControl` is a Software Task (i.e. a process to be mapped

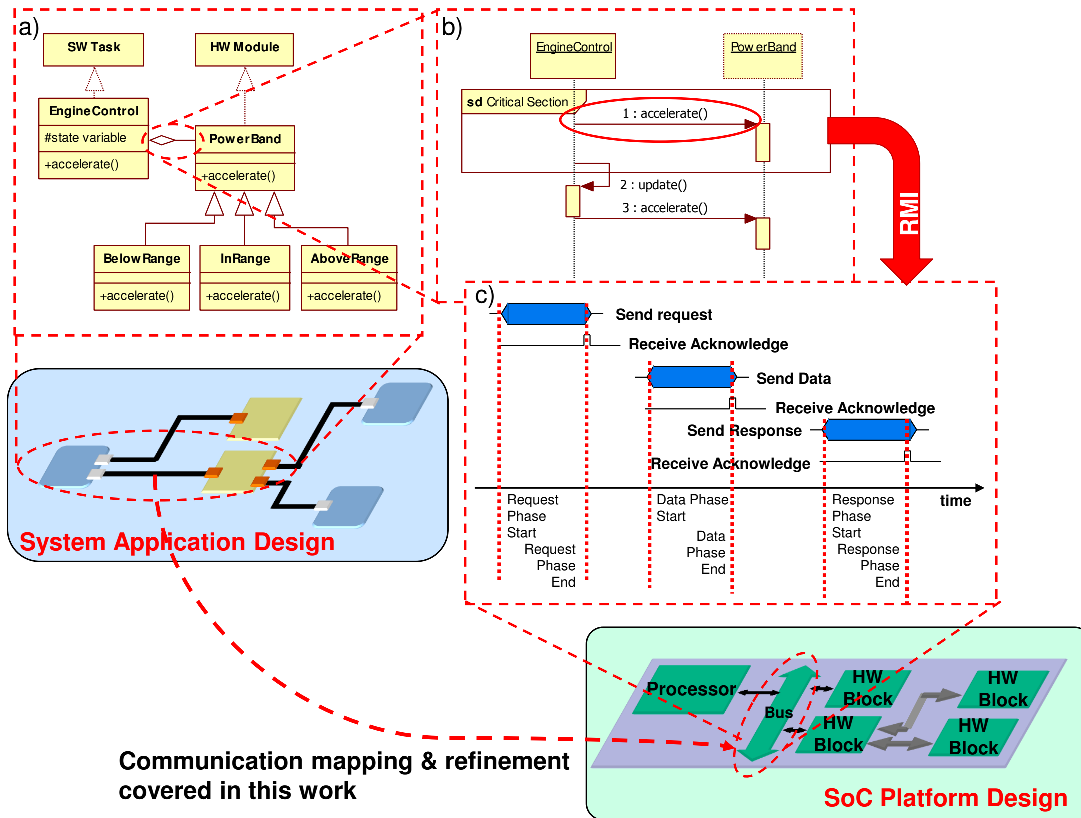


Figure 1.5: Basic Idea: Flexible Communication Mapping (simplified)

on a software processor of the target platform) and the `PowerBand` is a Hardware Module (i.e. a process to be realized as a custom hardware block in the target platform). This scenario describes a remote method call, since the accelerate method is called from one (software) process on another (hardware) process. Figure 1.5 b) shows the corresponding message sequence chart of this remote method call. On the target platform the accelerate method call is requested from the software processor and is being served by a custom hardware block. In this example the processor and the hardware block are connected by a shared bus. Figure 1.5 c) shows three basic protocol phases of a shared bus protocol: *Request*, *Data* and *Response*. The RMI protocol connects the method call request and invocation (Figure 1.5 b)) with the basic communication services of the target platform communication elements (Figure 1.5 c)).

Figure 1.6 depicts the concept for *system synthesis*, based on the virtual platform-based design approach presented in Figure 1.4. The idea is:

- To provide *Application Modeling Elements* Ⓐ consisting of Actors and Objects that encapsulate the functional behavior of the application and can be plugged together using *Ports* and *Interfaces*. A Port requires a certain service or set of services, while an Interface provides a service or a certain set of services. The Application Modeling Elements have an execution semantics, which can be implemented using a discrete event simulator. The Application Model can be executed and checked against a functional "golden model". Required timing properties of the application model can be expressed using so-called *Required Execution Time* (RET) blocks.

- To provide *Architecture Modeling Elements* Ⓑ consisting of processing, memory and communication elements which can also be connected through ports. These modeling elements implement services for the Application Model: Processing elements can execute Actors in hardware (i.e. as dedicated hardware) or software (i.e. as software on a processor) and share Objects between Actors, Memory Elements can store objects, and Communication Elements provide a Remote Method Invocation protocol stack to enable

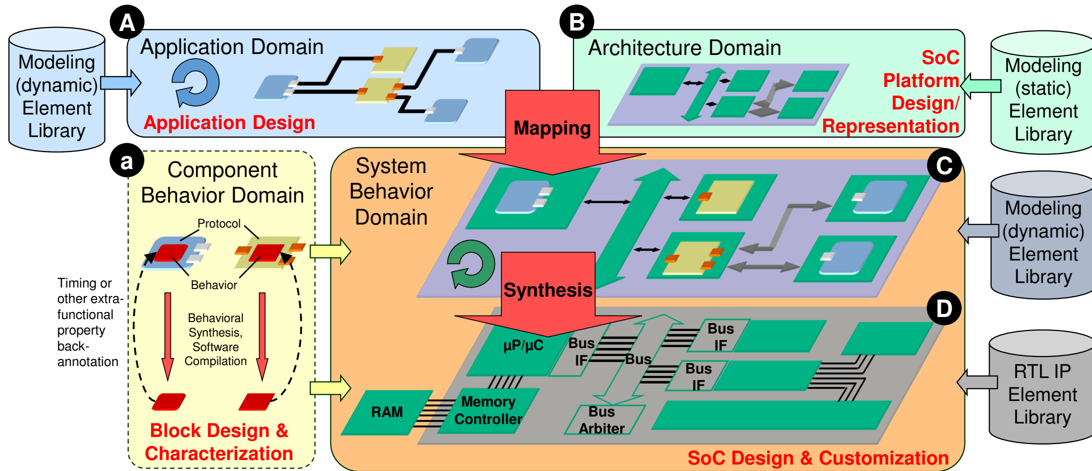


Figure 1.6: Basic Idea: System Synthesis

communication between Actors and Objects running on different processing elements (see Figure 1.5). The Architecture Modeling Elements have an execution semantics which can be implemented using a discrete event simulator, but they cannot be executed without an Application Model mapping.

- To map Application Modeling Elements to Architecture Modeling Elements Ⓒ, resulting in an executable virtual platform (called *Virtual Target Architecture Model*) of the overall embedded system (see Figure 1.4). Different instantiations and connections of Architecture Modeling Elements can be used to represent different target platforms. Pre-existing hardware IP components can be integrated using RMI wrapper objects (called sockets) which enable the translation of method calls to a pin-level protocol.
- To represent timing and other extra-functional properties of the RTL execution platform in the Virtual Target Architecture Model through back-annotation techniques ⓐ. The behavior, which is encapsulated by Actors and Objects, is refined to either the target platform using cross-compilation or behavioral synthesis techniques. The resulting timing properties of this refinement can be back annotated into the functional code of inside the Actors and Objects using so-called *Estimated Execution Time* (EET) block annotations. Communication and Memory Elements of the Architecture Model already contain target platform specific timing annotations. Finally, the mapping specific timing annotations can be checked against the RETs from the Application Model during model execution.
- To provide a synthesis semantics for each Architecture Modeling element for enabling a semi-automatic implementation of the Virtual Target Architecture Model as an SoC Ⓓ.

1.5 Outline

This work is structured in nine chapters and an appendix. The following chapters are organized as follows:

Chapter 2: Goals of the Methodology restricts the complex view of an embedded system to a manageable part which can be handled within the scope of the proposed methodology. The aim is to define the goals of the design methodology from a designer's point of view, especially the aspects concerning modeling, analysis, and synthesis of communication in an embedded SoCs. These goals will also serve for the evaluation of the proposed methodology in Chapter 5. The goals presented during this section have been extracted from a questionnaire which can be found in Appendix A.

The subsections of this chapter are organized as follows: Section 2.2 introduces the characteristics of embedded Systems on Chip that are in the focus of this thesis, followed

by Section 2.3 which gives an overview about certain strategies to implement communication in embedded HW/SW systems. Section 2.4 describes the goals on the design process with a focus on the communication between different components. Section 2.5 concludes this chapter and presents a tabular overview of all goals. The coverage of these goals will be checked in the conclusion of the subsequent chapters 5, 6, 7, 8 and 9.

Chapter 3: Terminology will define basic terms used in this thesis. It start with an overview of selected mathematical notations (see Section 3.2) used in this work. Since this work is about modeling, analysis and synthesis of embedded hardware/software systems, Section 3.4 introduces four different views on a model: Model of Computation (MoC), Model of Architecture (MoA), Model of Structure (MoS) and Model of Performance (MoP). These models are combined in the X-Chart and constrained for usage in this work.

Section 3.3 briefly introduces timed automata and UPPAAL for the specification of timed execution semantics and static analysis of the proposed modeling constructs.

An overview of different methodologies and a classification of the methodology proposed in this thesis is presented in Section 3.5 using the Y-Chart. This includes the presentation of a simulation-based design flow, and a general introduction to simulation and synthesis.

This chapter closes with a generic system-level design representation covering all relevant MoCs, enabling separation of computation and communication, used as a starting point for the proposed object-oriented extensions proposed in this thesis.

Chapter 4: Related Work starts with an overview of previous work used as background of the work presented in this thesis (Section 4.2). In the following sections, an overview and a classification of related work along the following topics is presented:

- Section 4.3: Object-oriented communication concepts in Electronic System-Level (ESL) design
- Section 4.4: SoC communication modeling
- Section 4.5: SoC communication synthesis
- Section 4.6: ESL synthesis methodologies

This chapter closes with a summary of contributions of this work (Section 4.7).

Chapter 5: Methodology, Modeling Elements and Operational Semantics presents the proposed object-oriented methodology, which is a combination of a platform-based design methodology, called system-level methodology, and an FPGA-based design methodology. For this purpose, certain combinations of Models of Computation, Architecture, Structure and Performance have been clustered in four different layers: *Behavioral*, *Application*, *Virtual Target Architecture* and *Implementation Layer*. In these different layers the associated design flow is described.

Section 5.2) gives a general overview of the abstraction layers supported by the methodology, from the pure functional design entry, down to the target platform implementation.

In Section 5.3 the general *Object Model* used in this thesis is described. In Section 5.4, Section 5.5 and Section 6.5 the Behavioral Layer, Application Layer and Virtual Target Architecture Layer modeling elements and their execution semantics is described.

After each of the detailed modeling element descriptions for the Application and the Virtual Target Architecture Layer in Section 5.5.2 and Section 5.6.2, mapping rules to establish the design flow for the transitions from one abstraction layer down to another are presented in Section 5.5.5 and Section 5.6.3. These sections also discuss the necessary design decisions to drive the mapping process. For the analysis of functional and extra-functional properties, simulative and analytic techniques can be used on the proposed abstraction layers. Simulative techniques will be described in more detail in Chapter 6, while static timing analysis techniques using timed automata is described in the operational semantics sections 5.4.3 and 5.6.4.

In Section 5.7 only a brief overview of the implementation on the Target Platform Layer is described. More details about the Virtual Target Architecture to Target Platform Layer mapping and synthesis step are described in Chapter 7.

This chapter closes with a summary and discussion of the requirements on the methodology and flow as demanded in Chapter 2.

Chapter 6: Simulation describes the simulation model of the modeling and refinement methodology presented in Chapter 5. The simulation covers the presented modeling elements of the *Behavioral*, *Application* and *Virtual Target Architecture Layer* and the specified mapping and refinement steps.

Section 6.2 gives a brief overview of SystemC and describes how OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) extends SystemC. Furthermore, the different layers of the OSSS simulation library are introduced.

In Section 6.3 the OSSS implementation of the Behavioral Layer is described. The OSSS Application Layer simulation model elements are described in Section 6.4 and the OSSS Virtual Target Architecture Layer simulation model elements are described in Section 6.5.

In Section 6.6 the mapping of an executable Application Layer model to a Virtual Target Architecture Layer model, including an example of simulative architecture exploration, is described.

This chapter closes with a summary of all supported simulation model elements and the implemented mapping relations between these modeling elements. The summary closes with a discussion of the requirements on the simulation or model execution as demanded in Chapter 2.

Chapter 7: Synthesis presents the OSSS synthesis flow that describes the semi-automatic transformation from an Application mapped to a Virtual Target Architecture to an FPGA target platform. The organization of this chapter is described along the implemented reference synthesis flow.

Section 7.2 gives a detailed overview of the entire OSSS synthesis flow. Section 7.3 describes the basic parsing and intermediate representation of the input model. Section 7.4 describes the representation of the Virtual Target Layer modeling elements for the chosen Xilinx FPGA platform. This is followed by the description of the platform synthesis process interfacing the specific Xilinx tools in Section 7.5. Section 7.6 describes the library-based software synthesis approach. Section 7.7 describes the custom hardware synthesis and Section 7.8 the Shared Object synthesis. Section 7.9 describes the integration with the Xilinx back-end synthesis and compilation tools to implement the specified system on the FPGA platform.

This chapter closes with a summary (Section 7.10) and a review of all synthesis related requirements from Chapter 2.

Chapter 8: Experiments contains experiments for the demonstration and evaluation of the presented methodology (see Chapter 5), simulation (see Chapter 6) and synthesis (see Chapter 7). The introduction of this section provides an overview of the presented experiments, metrics, covered requirements and links to the respective sub-section. This chapter closes with a discussion of requirements from Chapter 2.

Chapter 9: Conclusion concludes this work along the aims of this thesis from Section 1.3. The conclusion finally reviews all requirements from Chapter 2 and gives a summary of open issues and possible future work and research directions.

The appendices of this work contain additional optional material. For a brief overview, the appendices are organized as follows:

Appendix A: Survey contains a questionnaire that has been conducted in 2005 to obtain the "Requirements on Hardware/Software Communication Design based on Abstract Communication Models" [96] in the course of the ICODES project [223]. Participants in

this questionnaire have been three companies from the automotive, mobile communication and defense & security domain. The questionnaire has been used to derive requirements on the methodology of this work, as described in Chapter 2.

Appendix B: Timed Automata Templates and Examples contains additional information about understanding and reading Uppaal models, used to express the execution semantics of the modeling elements introduced in Chapter 5. Uppaal models are also used to demonstrate the analysis of safety and liveliness properties for the introduced object-oriented communication elements.

Appendix C: Pre-defined Shared Objects contains listings of pre-defined Shared Objects that replace *Shared Variables*, *Piped Variables*, *Queues Handshake* and *Double Handshake Channels* of the Behavior Layer.

Appendix E: Supported Target Platforms contains a description of supported target FPGAs, prototyping & development boards and basic IP components for the synthesis described in Chapter 7.

Appendix F: Synthesis Subset gives a short overview of the synthesizable SystemC/OSSS language constructs accepted by the current implementation of the *Fossy* synthesis tool, described in Chapter 7.

Appendix G: Integrated Development Environment gives an overview of the *Fossy* IDE, based on the Eclipse CDT Environment. It provides a tool suite for modeling, simulating, debugging and synthesis of OSSS and SystemC^{TM} designs.

Appendix H: OSSS Behavior Graphs contains plotted structure graphs of three different JPEG encoder models, described using OSSS Behavior Layer modeling elements. The corresponding experiment can be found in Section 8.2.

CHAPTER 2

Goals of the Methodology

2.1 Introduction

The main goal of the proposed methodology is to improve the design process of digital embedded hardware/software systems and in this context in particular to improve the design process of the communication between different system components.

Due to the wide variety of applications and their functional and extra-functional requirements, embedded systems are realized with many different technologies to meet the requirements of a certain type of application. These special characteristics of embedded systems make it difficult to define a single all-purpose design method. As a result the necessary skills and knowledge for embedded system design are as diverse as the systems that have to be built.

Therefore the concern of this chapter is to restrict the complex view of an embedded system to a manageable part which can be handled within the scope of the proposed methodology. The aim is to define the goals of the design methodology from a designer's point of view, especially the aspects concerning modeling, analysis, and synthesis of communication in an embedded SoCs. These goals will also serve for the evaluation of the proposed methodology in Chapter 5. The goals presented during this section have been extracted from a questionnaire which can be found in Appendix A. References to certain questions are indicated by Q#, for example Q33 for question number 33.

The following subsections are organized as follows: Section 2.2 introduces the characteristics of embedded Systems on Chip that are in the focus of this thesis, followed by Section 2.3 which gives an overview about certain strategies to implement communication in embedded HW/SW systems. Section 2.4 describes the goals of the design process with a focus on the communication between different components. Section 2.5 concludes this chapter and presents an overview of all goals.

2.2 Embedded Systems on a Chip (SoC)

In Section 1.1 we have given a general definition of embedded systems, their properties and, in particular, some properties of SoC platforms. This section aims for further restricting the set of embedded SoCs which can be covered by the proposed design methodology and synthesis technology.

The class of embedded HW/SW SoCs which will be considered during this thesis typically consist of software processors, hardware blocks (dedicated hardware accelerators, parallel and serial I/O components), memories, and interfaces to communication networks as shown in Figure 2.1.

[1] http://commons.wikimedia.org/wiki/File:ARMSoCBlockDiagram.png

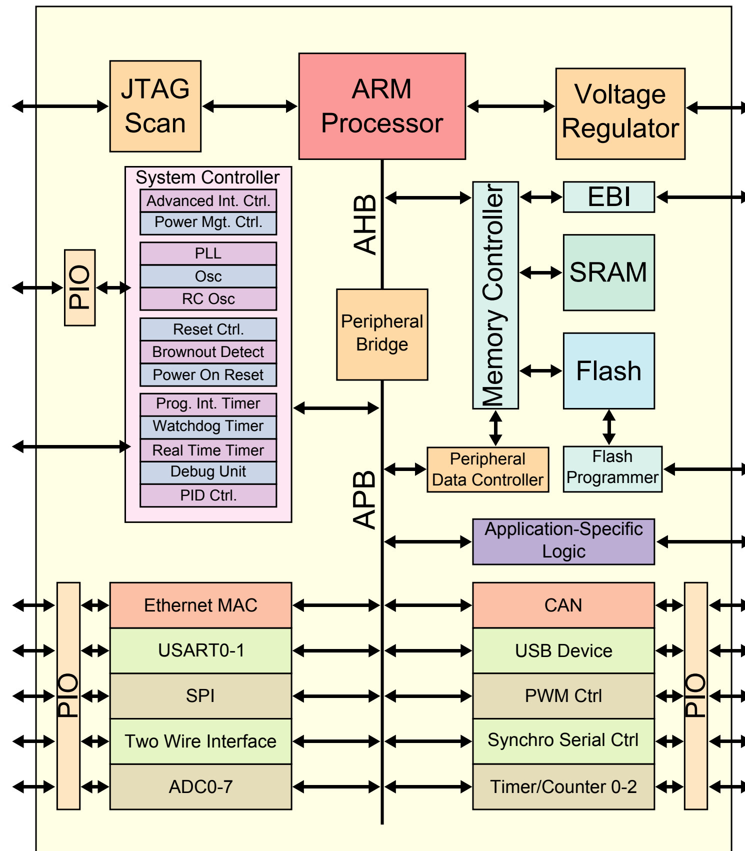


Figure 2.1: ARM processor based System on Chip[1]

2.2.1 IP components

With the ability of semiconductor manufacturing technology to integrate several billion transistors onto a single piece of silicon it is common to build the whole embedded system on a single chip (system on chip, SoC). Integrated micro-processors have been developed where the processor is combined with peripherals such as parallel and serial ports, DMA (direct memory access) controllers and interface logic to create devices that are already suitable for embedded systems by reducing the hardware design task and costs.

To optimize the design process of embedded systems with respect to development time and cost, the usage of IP (intellectual property) blocks became of significant importance.

IP blocks are available for many kinds of applications and can cover and affect the whole design process. IPs are mainly related to implement standard functions and standard interfaces. Typical IPs are microprocessors, buses, memories and I/O components (see Figure 2.1).

The results of the survey have shown that IP-cores are fundamental components to enable a fast development and will be used from 0 to 90% during the design process depending on the certain application which has to be realized.

2.2.2 Software Processors

Software processors are specialized hardware IP components which serve as software execution units. Each software processor is characterized by a single thread of control, apart from pseudo-parallelism introduced by (real-time) operating systems. The category of software processors includes off-the-self microprocessors, micro-controllers, DSPs and application specific processors (i.e. ASIPs).

All software processors suitable for the proposed methodology have to fulfill certain requirements. The processors should be able to communicate through a shared bus, and they should also be able to handle at least one external interrupt. Additional I/O ports and internal peripherals may be useful but not necessary. The processor's bus interface and interrupts are essential for building the communication interface to the communication network and thus other basic components (including other software processors, hardware blocks and memories).

Target architectures suitable for the proposed methodology should be able to support more than only one software processor in a single platform.

The questionnaire has shown that suitable software processors, with respect to the commercially available target architectures and target platforms, are:

1. hard core software processors: ARM, IBM PowerPC
2. soft core software processors: Altera Nios, Xilinx MicroBlaze

A soft core processor is a synthesizable component described in a hardware description language. Typically, the provider delivers the hardware description language code with additional synthesis scripts to ease the task of technology mapping, i.e. implementation for a particular technology process. Unlike soft core processors, hard core processors already exist as a physical layout or are even already mapped for a particular technology process.

Software can be seen as the implementation of an algorithm, which is usually a sequence of instructions. Since most processors have only one thread of execution, parallel behavior is not supported directly. Nevertheless it is possible to handle multiple concurrent tasks on one processor. This is done by sharing the processor's thread of execution among all tasks which are ready to run. This is called multitasking. It implicates the need for a scheduling algorithm. Depending on the complexity of the software either a customized scheduler can be designed or an operating system or real time operating system can be used. As the focus of this thesis is not on software modeling multitasking and operating systems are not supported in the presented methodology. An extension of the proposed methodology that enable modeling of parallel software systems using multitasking and real-time operating systems can be found in [48, 23, 17].

Embedded SoCs covered in this thesis share the important property of being designed for a specific application purpose. In contrast to general purpose computer systems like a desktop PC, the software part of an embedded system is generally fixed for the certain application.

The survey has shown that as source languages for the software in embedded systems often assembler, C and C++ are used. In the automotive area additionally synchronous languages like ESTEREL and the data flow language LUSTRE are used.

The entire methodology described in this thesis is implemented as an extension of the System Level Design Language SystemC. Technically, SystemC is a C++ class library, but from a methodical point-of-view it can be considered as a C++ extension. For this reason, modeling of software will be restricted to C/C++. Other software modeling languages can be used if they can be translated into C/C++ code.

2.2.3 Dedicated Hardware

Hardware blocks can either be pre-designed (i.e. IPs possibly with very restricted access and therefore only usable as a "black-box") or application specific and custom designed. Usually hardware blocks can be classified as soft macros (source code, netlist), hard macros (placed & routed "black-box" e.g. library element) or ASIC (which represents a chip as a whole). To connect these components with the system's communication network they must provide an appropriate communication interface. Hardware blocks which are capable of generating interrupts designated for a certain software processor or hardware blocks using DMA controllers possibly have to provide special interfaces to utilize these special communication mechanisms.

In this thesis we assume that application specific hardware is implemented either in a FPGA (Field Programmable Gate Array) or in an ASIC (Application Specific Integrated Circuit) or a combination of both. The questionnaire has shown that FPGAs are becoming more important. The reason for this is that nowadays FPGAs can handle the necessary design complexity. Furthermore, FPGAs combine the advantages of the flexibility of software through

their reconfigurability with the performance of hardware. However, FPGA implementations have some disadvantages such as higher power consumption and higher per unit cost for large volumes. Nevertheless, the support for FPGAs has no dramatic effect on the work within thesis, because modeling will be done on an abstract level, which is largely independent from the target technology. The final mapping to the desired target technology will be done by existing, state-of-the-art RTL synthesis tools.

The hardware is composed of several blocks of combinatorial logic and registers as well as wires connecting them. Therefore it is fully parallel, because the blocks can work independently from each other. The communication between the hardware blocks is done by signals. The considered hardware is synchronous that means it is triggered through a common clock signal which determines the timing behavior. Asynchronous hardware is not regarded in this thesis.

As the complexity of hardware was steadily growing, a more abstract mechanism was needed to design the hardware. Therefore hardware description languages in conjunction with synthesis tools are used in the design process. The most common hardware description languages are VHDL and Verilog. Both of them are are mainly used at RT level. SystemC builds on the foundations of VHDL and Verilog and will be used to describe custom hardware in this thesis. A SystemC behavioral RT to VHDL synthesis will be provided to target existing back-end RTL synthesis tools for custom hardware.

2.2.4 Memory

Memories are essential components in system design and can be treated as a special kind of pre-designed IP blocks. To use a memory block only its access scheme and access timing are required. A memory controller with an address decoder providing a communication interface for the communication network is used to access the memory from various system components. Software processors essentially need memories to store their data and program. Since hardware components might also heavily access memories for hardware/hardware or hardware/software communication, memories in embedded systems are often distributed instead of using a single monolithic memory.

2.2.5 Communication Interfaces

Communication interfaces provide links between components (that have been described in the sections above) and the communication network (described in the section below). These communication interfaces depend completely on the used communication network. Systems with a heterogeneous communication network, i.e. consisting of different hierarchical buses (with even possibly different bus protocols) connected by bridges, may include different communication interfaces dependent on which kind of bus the component is connected to.

The fact that there is no normalized communication interface for IP components often makes it necessary to build an adapter for the use with a specific communication network.

Components connected to a communication network can be classified in master (client) or slave (server) components with different communication interfaces. In a master-slave organization the master starts a transaction by issuing commands and the slave responds by sending/receiving data to/from the master. When multiple master components are connected to a communication network, arbitration has to be done in order to share the communication network among them. This necessary arbitration is done by an arbiter, which can be seen as a part of the communication network.

Regarding data transfers only, software processors are master components as they are usually used as central control units which contain the main thread of control. Software processors capable of handling interrupts can be regarded as both master and slave components whereas each interrupt handler provides some kind of software slave interface. DSPs and ASIPs as a special kind of software processors[2] can be both master and slave components as well. As DSPs or ASIPs are often used as co-processors to a general-purpose processor their behavior depends on the overall system control hierarchy (which can be classified from dependent co-processors over incrementally controlled and partly dependent co-processors to independent components)

[2] The DSP itself is equal to any other software processor besides specialization for a different kind of data and therefore provides a different instruction set.

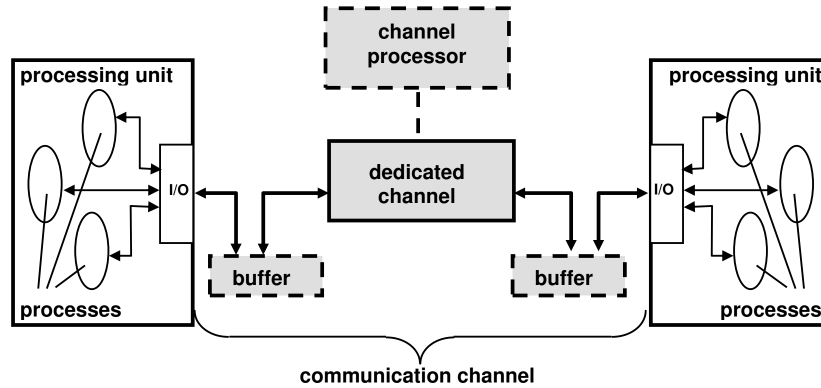


Figure 2.2: Point-to-Point communication [180]

[180]. Dedicated hardware blocks as mentioned above can be either master or slave or both master and slave dependent on their implemented functionality. Memories are always slave components as they never initiate any transaction by themselves.

2.2.6 Communication Networks

For SoCs a wide range of communication architectures like shared buses, crossbars, dedicated point-to-point connections, and network-on-chips (NoCs) exist. This section gives a short introduction to on-chip communication networks supported by the methodology presented in this thesis. In contrast to distributed embedded systems, where communication networks (i.e. Ethernet, LON, EIB, CAN, LIN, FlexRay, MOST etc.) are used to communicate over a comparatively large distance, on-chip communication networks addresses the inter-component communication over a shorter distance. NoCs are not addressed in this thesis.

At a high abstraction level communication is established through message passing on abstract channels. These abstract communication channels have to be mapped to physical on-chip communication networks as part of the synthesis process.

In the following a point-to-point, a shared bus, and a shared object communication network for SoCs will be presented.

2.2.6.1 Point-To-Point Communication

In a point-to-point communication (see Figure 2.2) two components are connected via a dedicated communication channel which optionally is buffered on each component's side. This buffering can be used to avoid active waiting and thus allowing the processing unit to proceed with its operation. The optional channel processor (or arbiter) has to ensure that each logical connection between processing units is assigned to a single dedicated physical channel at any time. In embedded SoCs this kind of point-to-point communication is often realized by a communication network called crossbar switch. In systems where only two components are connected and no connections to other components occur over time, the channel processor can be omitted and instead a hard-wired point-to-point connection can realize the communication network.

2.2.6.2 Bus Communication

On the one hand a point-to-point communication architecture achieves a great performance but on the other hand for huge designs many physical wires are necessary and this would lead to a big overhead in area consumption. Bus communication tries to overcome this disadvantage. A bus provides a physical channel which is shared among its connected components. Furthermore a bus provides a scheduling mechanism for shared medium accesses (as shown in Figure 2.3). This bus scheduling is usually done by an arbitration unit called bus-arbiter. Performance of bus communication can be increased by introducing hierarchical buses which are connected by bridges. Bridges are often used to separate the processor and the memory bus system from the bus system where the peripherals like timer and I/O components are connected. In this kind of

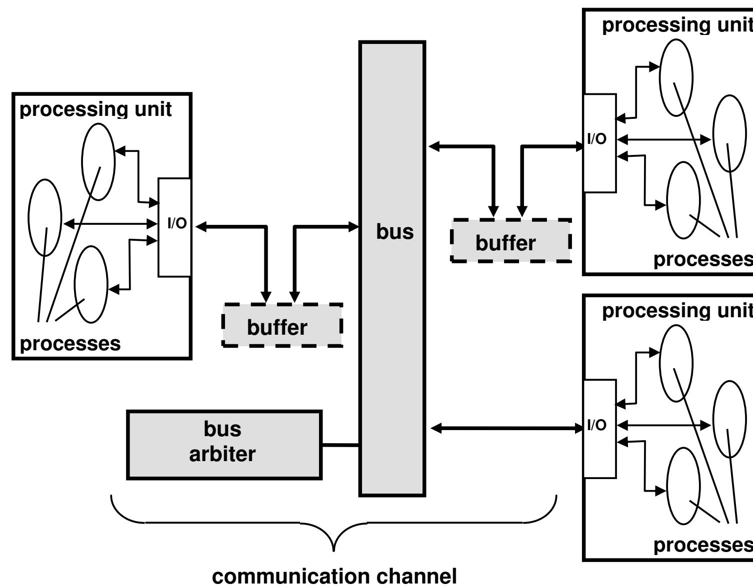


Figure 2.3: Shared bus communication [180]

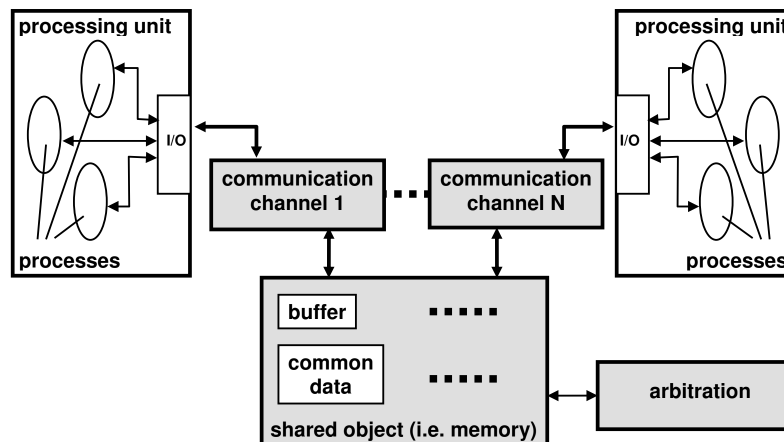


Figure 2.4: Shared object communication [180]

hierarchical bus systems the processor's bus operates at the processor's clock frequency and the peripheral bus operates at the peripheral component's clock frequency, which is usually slower than the processor clock.

As already described for point-to-point communication the communication interface of each component can be buffered optionally.

Typical bus systems used in SoC design are: ARM AMBA (Advanced Microcontroller Bus Architecture), IBM CoreConnect, Altera Avalon, and Silicore/Opencores Wishbone. A summary of these different bus systems can be found in [145, 84].

2.2.6.3 Shared Object Communication

Figure 2.4 shows shared object communication with the communication channel being either a point-to-point or bus communication architecture. The access to the shared object has to be controlled or arbitrated in order to guarantee mutual exclusive access to the data members it contains. The arbitration of the shared object can be realized in several different ways. Compared to the bus and point-to-point communication, the shared object communication is a more abstract concept. As a shared object can be seen as a slave component which responds to requests of master components, arbitration can either be done by the communication network it

is connected to, by the shared object itself (slave-side arbitration, in combination with point-to-point communication channels this is a cross-bar) or even a mixture of both. This kind of communication is very often used for shared memory communication where the shared object is represented by (or more precisely includes or encapsulates) a memory or a certain part of a physical memory. This kind of communication topology is mostly used to avoid multiple copies of the same data distributed to several memories.

2.2.7 Application mapping

A critical decision in embedded systems is to determine which parts of the design should be implemented in software and which in hardware. This is called partitioning. Generally, performance critical components are implemented in hardware while highly configurable or flexible parts should be implemented in software.

No matter how the system's functionality is split among the hardware and software, an essential part of the system is the cooperation of the partitions and consequently the communication, which will be discussed at Section 2.3 in more detail.

Through a HW/SW interface data, commands and events must be transferred across the HW/SW boundary. If a computation process is split into hardware and software, intermediate results have to be transferred for further computation. The interface is neither purely hardware nor purely software, but a mixture of both.

The HW/SW partitioning decision causes different communication scenarios. Therefore, a crucial task in system level design is the communication design and the interface synthesis. It is of great interest to explore as well the different communication alternatives to achieve an efficient and correct design and implementation of the interfaces.

This also becomes evident in case of modification or maintenance, where changes of hardware or software components very often entail changes in the interface. Such a modification may be a re-design of a component, changing its implementation from software to hardware or vice versa.

To summarize this subsection, we have discussed the properties of embedded systems which are of interest for the design methodology presented in this thesis. We have seen that hardware and software are both essential and tightly integrated parts of an embedded system. Therefore the design of an embedded system must equally consider both parts right from the beginning and throughout the whole design process.

2.3 Communication in Embedded System

One of this thesis' main issues is the design of the interfaces between hardware and software components of an embedded SoC. This section presents an overview of typical approaches which are common in the field of embedded HW/SW systems.

The different communicating processes either implemented in hardware (as components) or software tasks need a basic concept or mechanism for synchronization and transfer of data, as shown in Figure 2.5. In addition all communication partners must agree on the same interpretation of the exchanged data. For example the byte order (little endian and big endian), alignment and and data types (integer, floating point, ...) representation must be compatible.

In most cases the communication mechanisms are realized by protocols that implement handshake concepts, which handle the data transfer and the synchronization. These protocols are mapped to the interface primitives (communication units) provided by the SoC platform.

A communication protocol is a specified sequence of events (transitions) within given timing requirements that are needed to successfully transfer information and data on a logical or physical channel. An important criterion of a protocol is that parallel tasks have to synchronize their actions before they are able to communicate with each other, i.e., they have to perform some kind of handshake. For instance, if two processes want to communicate, both the sender process and the receiver process must be ready for the communication and they must be aware that the partner is also ready.

Since in most cases a shared medium (e.g. a bus) is used by several processes for the communication, the protocol needs a scheduling and arbitration scheme to determine the different accesses. For instance, if several processes use the same memory space it has to be

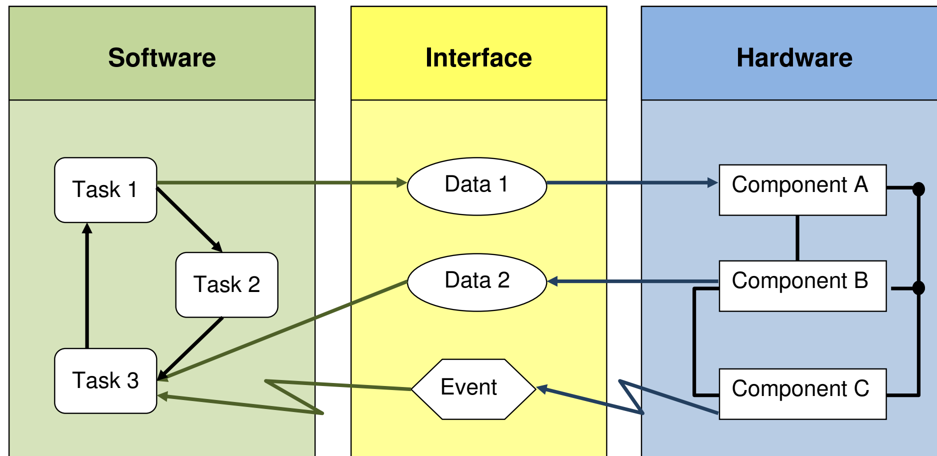


Figure 2.5: HW/SW communication in embedded systems (based on [111])

guaranteed that if a process is writing on the shared memory no other process is reading or writing. A mutual exclusion is necessary to avoid that more than one process uses the same data of the memory. To realize mutual exclusion several primitives are known for the software like semaphores, blocking, mutexes and critical sections.

The communication on the hardware platform can be distinguished in direct and indirect communication. The usage of direct communication means that the different components communicate directly over the I/O interfaces without the need of further glue logic components. The indirect communication indicates that the communication is more complex and the usage of glue logic like multiplexers, device registers, memories or special controllers is necessary.

Therefore, the demands on the HW/SW communication design are dependent on the chosen target platform, the processor type, the bus system, existing IP components and the desired protocol type. This includes the transfer mode, i.e. serial or parallel and which I/O standards should be used, the transfer type, i.e. synchronous or asynchronous and the available communication units.

The transfer type, i.e., synchronous or asynchronous bus transaction determines whether the protocol is defined with respect to the clock or whether a handshake protocol is required.

In a master-slave organization the master starts a bus transaction by issuing commands and the slave responds by sending/receiving data to/from the master.

2.3.1 Structural inter-component access techniques

To transfer data from the processor to an I/O device (or dedicated hardware) or vice versa, the processor must be able to address the device and supply one or more command words. There are basically two methods for this, namely memory mapped I/O and port I/O, which are described in the following paragraphs.

2.3.1.1 Memory mapped I/O

One traditional and most common interface design (method) is based on so called memory map tables denoting the mapping of shared data structures to the memory mapped I/O area. These maps describe in particular the memory address of control information and the access mode.

In memory mapped I/O, portions of the address space are assigned to I/O devices. Reads and writes to those addresses are interpreted as commands to the I/O device. If the processor places the address and the data on the memory bus, the memory system ignores the operation, because the address indicates a portion of the memory space used for I/O. The device controller recognizes the operation and transmits the data to the device.

For the realization of this method the software designer encapsulates the memory mapping in functions to access the shared variables and the hardware designer in a VHDL package. The

encoding and decoding of data types like large integer into bit vectors which fit into the shared registers has in the most cases to be done manually.

2.3.1.2 Port I/O

Another method for the communication is the usage of dedicated I/O ports of a processor. The processor provides special instructions for accessing these ports and optionally instructions for configuring them as inputs or outputs.

2.3.2 Behavioral inter-component access techniques

In order to control the inter-component data transfer, which can be regarded as a sequence of events specified by a certain protocol, there are two different mechanisms: *polling* and *interrupts*. The data transfer itself can be realized by a software processor by a DMA-Controller or any other bus master component.

These different mechanisms have different advantages and disadvantages concerning the processor utilization, the performance and the costs of the I/O system. For this reason an appropriate solution concerning the desired application has to be chosen. Nevertheless, the target platform has to support some of the presented mechanisms explicitly in order to control and realize inter-component data transfer.

2.3.2.1 Polling

Polling is a mechanism in which one component actively checks another component whether an action is required or if a computation has been finished and the results are available. In most cases a software component periodically reads status registers of certain custom hardware or I/O device to check whether an operation is necessary. Hence, the main task is periodically suspended by the periodic checks.

Polling by its very nature is going to find that no action is required more often than it will find that servicing is needed. If this is not the case, data could be lost due to under-sampling which might lead to data over- and under-runs. This means polling spends a lot of time in non-constructive work. In many embedded systems this is not a problem, but in low power systems, for example, this unnecessary processing and power consumption might not be acceptable.

If the processor does not perform any other actions than polling, for example because it cannot proceed with its task without the result it is waiting for, this is called *busy waiting*.

Polling is also used in hardware to check for incoming data or events. Despite the extensive parallelism of hardware busy waiting can induce the same negative effects as it has in sequential software programs. Concerning inter module communication the feasibility of polling depends on the used hardware/hardware communication network. While polling is no problem when using point-to-point communication only, it has a major drawback when using shared buses. Polling easily "pollutes" the bus, which results in a waste of bandwidth. The periodic accesses to the polled registers also reduce the possible access rate for other module.

To realize hardware/software communication through polling no special hardware components are required. Polling is supported by any target architecture meeting the requirements above.

2.3.2.2 Interrupts

Interrupts are another way to communicate and synchronize. Interrupts can appear at any time and are not predictable. Therefore they represent an asynchronous communication mechanism. The architecture of the embedded system defines the available interrupt mechanism which can be used by the designer. Typically, an interrupt is sent from a hardware component to one of the software processors to request servicing. Interrupts alone just carry the information that an event has occurred. The software part of the application running on the processor handles the recognized interrupt by so called interrupt service routines (ISR) or interrupt handlers. To each possible interrupt an associated interrupt handler has to be provided that reacts to the certain interrupt. The usage of interrupts can be necessary in two cases: On the one hand if an event has occurred in the hardware or the environment that requires an immediate activity in the software (ISR has to intervene immediately, possibly complex ISR behavior), on the other

hand if the processor has to be informed about the completion of a certain task on a specific hardware component (ISR sets a flag in software, normally simple ISR behavior). Interrupts can also be used to send events from one task running on a software processor to another task running on another software processor.

After an interrupt from the hardware (or another processor) is recognized, the associated interrupt service routine suspends the current running task on the processor and decides in which state the system should continue after the execution activated by the interrupt is completed.

If several interrupts are supported by the processor and more than one is active at the same time, the question is which interrupt has to be served first. One way is to handle the interrupts by the order in which they have been activated, such that the first interrupt will be served first and all others will be blocked for that time. This way is applicable for interrupts without priority and where a short time is necessary to handle the interrupts, including the recognition, the processing and the return.

Another solution is that the interrupts convey further information like the priority. Consequentially, the priority associated with the interrupts is sufficient to decide which interrupt handler has to be executed first if more than one interrupt is activated at the same time.

To realize hardware/software communication through interrupts the target architecture has to fulfill certain requirements. First of all the used software processors must be able to process at least one interrupt, providing a context switching mechanism and an input port for the interrupt line. When using multiple interrupt sources with only limited interrupt ports at the processor, the use of an interrupt controller becomes essential. The interrupt controller is either connected to the software or to the hardware/software communication network as a slave component. Additionally it has dedicated unidirectional point-to-point connections to the interrupt ports of each software processor. An interrupt controller should at least provide masking and prioritization of interrupts. The advantage of an interrupt driven communication is that an interrupt eliminates the need for the processor to poll devices and instead allows the processor to focus on executing the main application. Furthermore, interrupts usually greatly reduce the latency time between an event is issued and the event is handled.

Since interrupts are suspending the current application running on the processor, many interrupts can prevent the completion of the main task in an adequate time interval.

Furthermore, with high speed ports, which in turn cause high interrupt frequencies, the cost of interrupting the processor can be higher than the execution to empty or load a buffer. In these cases, the limiting factor for the data transfer is the time needed to recognize, to process and to return from the interrupt. In these situations it is usually more efficient to store the data locally in a queue/memory and to handle the data transfer via DMA.

2.3.2.3 DMA

Direct memory access (DMA) is a concept to relieve the processor from pure data transfers between basic system components (hardware block $\leftrightarrow$ hardware block, hardware block $\leftrightarrow$ memory, memory $\leftrightarrow$ memory) and allows it to concentrate on its main task rather than on the transfer of data.

In the following a DMA-Controller can be understood as a hardware component which is capable to access a memory or any other peripheral component without occupying the software processor. Nevertheless, this basic principle is applied in different ways. Thus there are systems with a central DMA-Coprocessor, but also systems in which every component has its own decentralized DMA controller.

DMA can be used to reduce the need to interrupt the processor because interrupts are only issued after a DMA transfer is completed. Based on the assumption that a DMA transfer encapsulates a sequence of N data transfers the need to interrupt the processor can be reduced to 1/N compared to a pure PIO[3] transfer. This reduction of the interrupt rate can also be achieved when using bigger buffers and generating an interrupt when the buffer is full. But nevertheless without using DMA the processor is still occupied with copying data.

To use DMA in conjunction with the communication network of a chosen target architecture either special hardware components (DMA controller) or special communication interfaces for

[3] Programmed Input/Output where an interrupt is issued after each data word is available for copying by a processor

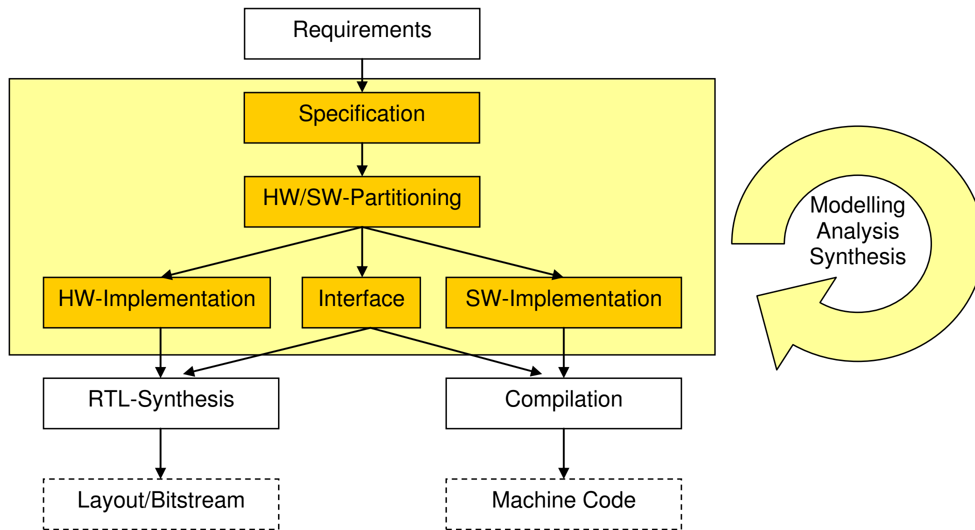


Figure 2.6: Overview of the general design flow and the parts covered in this thesis

each hardware component capable of DMA (in a shared bus architecture these communication interfaces are classified as bus master or master-slave components) have to be provided. Each master component issuing a direct memory access requests for the communication network (e.g. the bus), addresses it and writes directly to the connected memory or other slave peripheral. In this case some kind of DMA controller is implemented by the hardware component's master interface. Generally a DMA controller can be regarded as a special hardware component (master) connected to the systems communication network capable of issuing data transfers to other components (slaves).

The DMA controller is a device that can move data between dedicated hardware (i.e. special I/O components) and memory areas or between memory areas of single or possibly different physical memories. The DMA controller acts as a hardware implementation of the low-level buffer filling or emptying interrupt routine. It is preferably used for high bandwidth devices where the transfer consists primarily of relatively large blocks of data.

For DMA execution the controller needs some parameters that define the data transfer. One parameter is to determine the memory address which has to be written to or which has to be read from. Another one specifies the size of the data package to be transferred. The last parameter encapsulates the information which device has to be informed after the data transfer is completed.

2.4 Requirements on Communication-Centric Design

This section will describe the requirements on the design process with a focus on the communication between different components. Figure 2.6 depicts a general overview of the proposed design flow and highlights the parts covered in this thesis. These include the specification phase and HW/SW partitioning. For the HW/SW partitioned system description an implementation of the hardware, software and interface part shall be derived.

For all phases covered in this work, modeling and analysis will be supported. Regarding synthesis, the specification phase is considered to be covered only partly, because the initial executable models, written by the designer, are most probably not synthesis able. After manual refinement following the presented design rules however, the aim is to have an executable specification of the system including hardware and software parts. This HW/SW partitioned model will then be further processed automatically through synthesis. The synthesizer generates the appropriate output language for the target software and hardware parts of the design.

Synthesizable in the context of this thesis relates to the synthesis subset as defined in Appendix F. The design process from *RTL-Synthesis*, *Software Compilation* and below is already

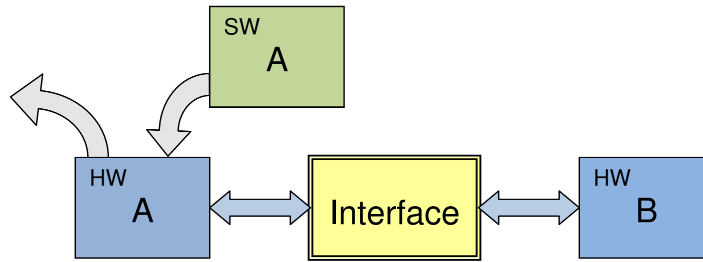


Figure 2.7: Explicit interface objects facilitate component interchangeability

state of the art and not in the focus of this thesis. However, it is absolutely crucial to ensure that the proposed methodology seamlessly builds on top of existing synthesis technologies in order to assure a continuous design flow.

The requirements mentioned in the following sections are based on the questionnaire in Appendix A and are structured according to the design process activities: *Modeling*, *Analysis*, and *Synthesis*. In the following sections we will discuss the different requirements from the questionnaire. This chapter closes with a tabular summary of the requirements.

2.4.1 Modeling

The methodology shall provide a modeling language which is able to express hardware and software components and especially the communication between these (Q44) in a single model. The modeling language which will be based on SystemC must be able to allow executable specifications on the one hand and synthesizable specifications on the other hand. The simulation, i.e. the execution of the specification, shall reflect the system behavior including hardware, software and their interaction.

As can be seen from the questionnaire (Q4) Matlab/Simulink and C/C++ are very popular modeling language for the initial specification. In this thesis a Matlab/Simulink entry will not be supported. But anyhow, using a C code generator for Matlab/Simulink models (e.g. Simulink Coder aka Real-Time Workshop) enables a manual integration of Matlab/Simulink models in the supported SystemC-based specification model. Without major manual recoding support this Matlab/Simulink specification block can only be mapped to Software. Regarding existing C/C++ models, the same restriction applies. Without manual recoding effort existing C/C++ models can only be mapped to software. Technically, the integration of C/C++ models in our specification model is straight forward, since SystemC is a C++ library.

In order to show the timing behavior of the HW/SW system, the underlying model needs a notion of time. The methodology should be able to cover untimed (purely functional) models, transaction-level models and cycle accurate models, but not all of these models need to be synthesizable.

The communication between different components plays a central role, due to its importance in embedded systems as described in Section 2.3. Hence, the methodology needs to provide modeling elements which allow to describe the communication in such a way that a hardware implementation of a component can be exchanged by an equivalent software implementation without changing the rest of the system - or at least with minimal changes. In other words, the methodology should allow an easy repartitioning of the design. Easy repartitioning is preferable for the evaluation process and for the usage of IP blocks which may have to be changed. Therefore the methodology shall provide constructs for a uniform interface description (Q19) of hardware and software components. This uniform interface is the envisioned mechanism to allow the interchangeability of components as shown in Figure 2.7.

Besides the requirement of interface objects to facilitate a common handling of software and hardware components they shall also provide modeling constructs for abstract communication. For example, method-based or transaction-level interfaces are envisioned instead of signal-level interfaces as used in VHDL and Verilog.

Such an envisioned mechanism allowing a common interface of hardware and software components as well as an abstract communication modeling will be based on the concepts of the so called *Shared Objects*. These Shared Objects were introduced and used in the former work for abstract communication (see Section 4.2). Up to now the shared objects have been limited to HW/HW communication and need to be extended and modified to handle HW/SW communication as well.

For the hardware modules it is necessary to be still able to describe the modules at RT-level including constructs like generating for-loops (Q16). Note that this requirement partly conflicts with requirement of abstract, HW/SW-independent interfaces. For further discussion see Section 2.5.

Concerning the software, the two most important topics are support of RTOSs (real time operating system) and multitasking. For these two topics the user requirements diverge, especially the RTOS support ranges from not necessary to mandatory (Q26). Common, however, is the growing importance of multitasking and the need to consider timing properties (real time) of the software (Q27). From the modeling point of view it seems to be most advantageous to keep the abstract models free from RTOS specifics and to use generic multitasking constructs instead.

Since a design is usually not completely written from scratch a very important issue is the integration of IP components (Q7). Basically there are two classes of IP components: general purpose components like CPUs, microcontrollers, DSPs, Memories, etc. and application specific hardware components like special signal processing blocks, e.g. special filters (Q8), which have to be regarded.

2.4.2 Analysis

Analysis in the context of this work means execution of the specification and has basically two purposes (Q5). First, it is done for functional exploration during algorithm development. Second, it is done for functional verification in order to assure that functional correctness is preserved during the design process. For this purpose debugging needs to be supported at all levels of abstraction in the design process (Q33).

There is work in the area of *Design Space Exploration* (DSE) [5] that explicitly deals with analysis and (semi-)automatic optimization of further metrics that go beyond the ones mentioned here. Existing techniques from this research area can be used to guide the designer to find the optimal architecture and HW/SW partitioning under the given constraints. This will be an improvement with respect to the current decision process, which is based either on expert advice/experience or even fixed by a predetermined architecture (Q6).

Currently the functional verification of the system is mainly done by executing the application on a prototypical hardware or a special development board (Q31). The proposed methodology shall facilitate a system simulation, i.e. hardware and software, with a reasonably high simulation performance even for complex systems. It shall be possible to run the system simulation without, i.e. before, the actual target hardware is available or even determined. The simulation must reflect the functional behavior of the system as well as basic timing properties of the potential architecture it will be mapped onto.

Ideally it should be possible to combine models of different levels of abstraction in a single simulation. For example, combining a clock cycle accurate RTL (register transfer level) hardware block with an untimed algorithmic block and a software task. This feature will help to establish a stepwise and incremental refinement process (Q18, Q19, Q20).

Finally, the integration of IP components, which was also a requirement for the modeling phase, has to be considered for the simulation. Therefore it is necessary for the methodology to provide concepts to integrate the IP components into the system simulation.

2.4.3 Synthesis

The methodology shall be supported by a (prototypical) synthesis tool. As mentioned before, synthesis within the context of this work refers to the transformation of the HW/SW partitioned model into a hardware, software and interface implementation, instead of classical behavioral synthesis, which is also often called high-level synthesis. Synthesis within the context of this work will process the HW/SW partitioned system model and produce C/C++ code for the

software part (Q35) and VHDL for the hardware part (Q34). The languages ESTEREL and LUSTRE were also mentioned (Q28), but will not be supported in this work. The generated software code shall be compliant with C++ standard (ISO/IEC 14882:1998) that it can be processed with common C++ compilers (e.g. gcc). The generated VHDL code for the hardware shall be compliant with the synthesizable subsets of Synopsys Design Compiler and Synplify Pro from Synplicity (Q37). Furthermore, the generated code has to be readable for a human being such that it is possible to trace problems, bottlenecks or bugs back to the input source code construct which was the cause of the problem (Q36).

One of the synthesis' main tasks is to map the abstract communication objects onto concrete mechanisms such as memory mapped I/O shared memory, interrupts, polling, DMA or proprietary direct HW/HW communication (Q38, Q39, Q42) and to generate the necessary hardware and software parts. Especially for the integration of IP components it is necessary that the designer can control the synthesis in order to enforce a certain communication mechanism, which is required by the IP component. This can be done for example by constraints in the synthesis script or by special statements within the source code.

The target technology on which the abstract model can be mapped is described in Section 7.4. The supported hardware platforms are FPGAs from Xilinx along with its on-chip CPUs (PowerPC and MicroBlaze) and its on-chip bus systems (Q43). This thesis will not cover RTOS support for the software part (Q41), extension of this work to support RTOS can be found in [48, 23, 17].

A further requirement for the code generated by the synthesis tool is efficiency (Q32) in terms of area and critical path (determining $f_{\max}$, the maximum clock frequency).

2.4.4 Implicit Requirements and Consequences

As described in Section 2.4, SystemC will be used as system modeling language for the software, the hardware, and the interfaces in between. Although SystemC is based on C++, which is a software language, SystemC lacks a concept to handle the description of embedded software. As mentioned before, the simulation kernel handles the progress of time, which is only appropriate for hardware descriptions. The simulated time advances only at `wait()` statements and all other statements do not consume any time. Traditional C++ does not include any explicit information about the run-time behavior. Therefore simple calls of C++ code within a SystemC model implicitly assume a zero execution time. Whereas this is sufficient for a pure functional simulation, it is not sufficient for simulating the interaction between hardware and software. Hence it is necessary to develop and introduce a notion of time for the software parts of the model as well. A major concern is the trade-off between accuracy and simulation performance. For example, a cycle accurate model of the processor including detailed cache behavior will give very detailed and accurate results, but depending on the model complexity a system simulation may take too long.

Due to the different nature of Hardware and Software descriptions there will be two different coding styles, i.e. synthesis subsets: one for the hardware modules and one for the software modules.

The analysis of the questionnaire has shown that there is a wide variety of IP blocks which have to be regarded. Within the context of the this thesis, these IP blocks can be classified into four categories. The first category comprises complete software processing elements, whereas the second category comprises application specific hardware processing elements. The third category comprises memory elements, and the fourth category comprises communication routing infrastructure (e.g. bus, point-to-point, cross-bar, ...). The proposed design methodology needs to define how the different views (functional, structural) on theses IP components shall be reflected at the different design phases.

The behavior of application specific IP blocks has to be reflected also at the abstract modeling level, whereas software processor IP blocks do not have an explicit representation in the abstract model, but only their influence on the timing of the behavior running on them might be of interest. IP blocks are becoming even more important during the synthesis phase, because the abstract model will partly be mapped onto them. For synthesis the integration of existing VHDL and Verilog IP models shall be possible.

Considering system simulation, the main conflicting topics are the same as for the software simulation, namely accuracy and simulation performance. To mitigate the problem it should be

possible to simulate parts of the design on different levels of abstraction. Modules on different levels of abstraction have different kinds of interfaces, for example method based communication on the abstract level versus signal based communication on RT level. Therefore a kind of adapter is needed for the simulation to connect the model on the lower level of abstraction to the one which is on the higher level of abstraction. In the case of an abstract (high level) model being synthesized to a lower level implementation, the adapter could be generated by the synthesizer, too. However, if the designer prefers to describe some of the components directly on RT level, the adapter to the abstract interface has to be created manually.

2.5 Summary

In Section 2.2 and Section 2.3 the most important aspects of embedded systems with respect to the work in this thesis have been described. Section 2.4 gave the detailed goals which were derived from the survey. This conclusion summarizes the goals of the methodology, as presented in Table 2.1. Most of the goals have been derived from the questionnaire which can be found in Appendix A. References to certain questions are indicated by Q# (e.g. Q33 for question number 33). Goals that have not been obtained from the questionnaire are marked as "self".

Table 2.1: Summary of the goals of the methodology (G: general, M: modeling, A: analysis, S: synthesis)

ID	Goals	Source
G1	Integration of synthesis tool and simulation infrastructure into Eclipse CDT Framework	self
G2	Introduce a notion of time for the SW parts	self
M1	Single modeling language to describe HW and SW	Q18
M2	SystemC approach	self
M3	Executable Specification and HW/SW partitioned models	Q4, Q5
M4	Synthesizable HW/SW partitioned model	self
M5	To be able to cover untimed (purely functional) models, transaction-level models and cycle accurate models	Q20
M6	Methodology needs to provide modeling elements which allow to describe the communication	Q19, Q44
M7	Easy HW/SW repartitioning of the design (a SW module can be replaced by a HW module without manually modifying its communication interfaces)	self
M8	Provide constructs for a uniform interface description	Q19
M9	Provide modeling constructs for abstract communication	self
M10	Possibility to write hardware modules at RT-level	Q16
M11	Support of multitasking	Q26
M12	Consideration of (real-)time constraints	Q27
M13	Support of operating systems	Q41
M14	Integration of IP components	Q7, Q9
A1	Debugging on all levels of abstraction	Q33
A2	High simulation performance (at least higher that state-of-the-art RTL simulations)	Q11, Q14, Q17
A3	Basic timing properties shall be reflected by the simulation	self

continued on next page

Table 2.1: Summary of the goals of the methodology (G: general, M: modeling, A: analysis, S: synthesis) (continued)

ID	Goals	Source
A4	Combine models of different levels of abstraction in a single simulation	Q18, Q19, Q20
A5	Consideration of IP components in the simulation	Q7
S1	Provide a (prototypical) synthesis tool	self
S2	Software output language C++ compliant with C++ standard (ISO/IEC 14882:1998)	Q35
S3	Hardware language VHDL compliant with the synthesizable subsets of Synopsys Design Compiler and Synplify Pro from Synplicity	Q34
S4	The generated code has to be readable for a human being	Q36
S5	Possibility to map the abstract communication objects onto concrete mechanisms such as memory mapped IO/ shared memory (using polling, interrupts and/or DMA) or proprietary direct HW/HW communication and to generate the necessary HW and SW parts	Q38, Q39, Q42
S6	For the integration of IP components it is necessary that the designer can control the synthesis and to enforce a certain communication mechanism, which is required by the IP component	Q7, Q10
S7	Control of the synthesis by constraints in the synthesis script or by special statements within the source code	Q44
S8	Efficiency of the generated code (for hardware: area and critical path; for software: memory footprint) compared to a hand-crafted design	self

CHAPTER 3

Terminology

3.1 Introduction

The chapter will define basic terms used in this thesis. Most of these terms are overloaded and ambiguously used in different areas of computer science. To avoid any ambiguities this section will provide a brief definition and describe where in this thesis the definition is required.

This chapter starts with an overview of selected mathematical notations (see Section 3.2) used in this work. Since this work is about modeling, analysis and synthesis of embedded hardware/software systems, Section 3.4 introduces four different views on a model: Model of Computation (MoC), Model of Architecture (MoA), Model of Structure (MoS) and Model of Performance (MoP). These models are combined in the X-Chart and constrained for usage in this work.

Section 3.3 briefly introduces timed automata and UPPAAL for the specification of timed execution semantics and static analysis of the proposed modeling constructs.

An overview different methodologies and a classification of the methodology proposed in this thesis is presented in Section 3.5. This includes the presentation of a simulation-based design flow, an introduction to simulation and synthesis. The different design methodologies are presented using the Y-Chart.

This chapter closes with a generic system-level design representation covering all relevant MoCs, enabling separation of computation and communication, used as a starting point for the proposed object-oriented extensions proposed in this thesis.

3.2 Selected Mathematical Notations

This section gives and overview and short explanation of the mathematical notations used in this thesis.

Definition 3.2.0.1 (Formulas or predicates):

$$
\begin{aligned}
\boldsymbol{T} \quad & true \\
\boldsymbol{F} \quad & false \\
\neg P \quad & not\ P \\
P \vee Q \quad & P\ or\ Q \\
P \wedge Q \quad & P\ and\ Q \\
P \Rightarrow Q \quad & P\ implies\ Q \\
P = Q \quad & P\ equals\ Q
\end{aligned}
$$

$$\begin{array}{rl}
\forall x\colon P & \textit{for all } x, P \textit{ holds} \\
\exists x\colon P & \textit{there exists } x \textit{ such that } P \textit{ holds} \\
\exists!\, x\colon P & \textit{there exists exactly one } x \textit{ such that } P \textit{ holds} \\
\forall x \in A\colon P & \textit{for all } x \textit{ in } A, P \textit{ holds} \\
\exists x \in A\colon P & \textit{there exists } x \textit{ in } A \textit{ such that } P \textit{ holds} \\
\exists!\, x \in A\colon P & \textit{there exists exactly one } x \textit{ in } A \textit{ such that } P \textit{ holds} \\
x \in A & x \textit{ is an element of set } A
\end{array}$$

□

Definition 3.2.0.2 (Sets):

$$\begin{array}{rl}
\{x \mid P\} & \textit{the set of all } x \textit{ that satisfy } P \\
A \cup B & \textit{the union of sets } A \textit{ and } B \\
A \cap B & \textit{the intersection of sets } A \textit{ and } B
\end{array}$$

□

Definition 3.2.0.3 (Tuples):
For an arbitrary domain D and for elements $d_1, d_2, \ldots, d_n \in D$, the according n-tuple is written in the following notations:

unrolled: $(d_1, d_2, \ldots, d_n)$

indexed: $(d_i)_{i\in\{1,\ldots,n\}}$

The domain of tuples of length n is written $D^{1,n}$:

$$D^{1,n} \quad := \quad \{(d_i)_{i\in\{1,\ldots,n\}} \,|\, d_i \in D\}$$

The empty tuple will be written ϵ.
The length of tuples of arbitrary domains D can be obtained using $|\cdot|$ metrics:

$$\begin{array}{rcl}
|D^{1,n}| & = & n \\
|\epsilon| & = & 0
\end{array}$$

The domain of vectors of length n is written $\overline{D}^n$, where $\overline{D}$ denotes vectors of dynamic length:

$$\overline{D}^n \quad := \quad \left\{ \begin{pmatrix} d_1 \\ \vdots \\ d_n \end{pmatrix} \,|\, d_i \in D \right\}$$

The length of vectors can also be obtained using the $|\cdot|$ metrics:

$$\begin{array}{rcl}
|\overline{D}^n| & = & n \;:\; \textit{for static vectors} \\
|\overline{D}| & = & m,\; \textit{with } m \in \mathbb{N}_{\geq 0} \;:\; \textit{for dynamic vectors}
\end{array}$$

□

Definition 3.2.0.4 (Kleene Closure):
Given an arbitrary domain D, we define:

$$\begin{array}{rcl}
D^+ & = & \bigcup_{n\in\mathbb{N}} D^{1,n} \\
D^* & = & D^+ \cup \{\epsilon\}
\end{array}$$

□

Definition 3.2.0.5 (Powerset):
For an arbitrary domain D, let $\wp(D)$ be its powerset, i.e., the set of all subsets of D.

$$\wp(D) := \{D' \mid D' \subseteq D\}$$

□

Definition 3.2.0.6 (Image):
Given a function $f : M \to N$ and a set $M' \subseteq M$, the image of M' will be written $f(M')$ and is defined as follows:

$$f(M') := \{f(m) \mid m \in M'\}$$

The image of f is a special case $f(M)$. □

3.3 Timed Automata

This section provides the basics of timed automata [186] used to describe the operational semantics of the proposed modeling language in Chapter 5.

3.3.1 Definition

A timed-automaton is a finite-state machine extended with clock variables. It uses a dense time model where a clock variable evaluates to a real number. All clocks progress synchronously. A system can be modeled as a network of timed atuomata in parallel. For improving the applicability of timed automata the basic timed atuomata model is further extended with bounded discrete variables that are part of the state. These variables can be used as in imperative programming languages. Variables can be read, written, and are subject to arithmetic operations. A state of such an extended timed automata system is defined by the locations of all automata, the clock values, and the values of the discrete variables. Every automaton may fire an edge (or transition) separately or synchronize with other automaton, leading to a new state.

We use the following notations: C is a set of clocks and $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie y$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed-automaton is a finite directed graph annotated with conditions over and resets of non-negative real valued clocks.

Definition 3.3.1.1 (Timed Automaton (TA)):
A timed automaton is a tuple $\mathcal{A} = (L, l_0, C, A, E, I)$, where

- *L is a set of locations,*
- *$l_0 \in L$ is the initial location,*
- *C is the set of clocks,*
- *A is a set of actions, co-actions (synchronization with other automata through channels. $c!$ for notification of an event on channel c and $c?$ for the reception of an event on channel c) and the internal τ-action,*
- *$E \subseteq L \times A \times B(c) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and*
- *$I : L \to B(C)$ assigns invariants to locations.*

□

A clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. Guards and invariants are considered as set of clock valuations, writing $u \in I(l)$ meaning that u satisfies $I(l)$.

Definition 3.3.1.2 (Semantics of TA):
Let (L, l_0, C, A, E, I) be a timed automaton. The semantics is defined as a labeled transition system $\langle S, s_0, \to\rangle$, where

- *$S \subseteq L \times \mathbb{R}^C$ is the set of states,*
- *$s_0 = (l_0, u_0)$ is the initial state, and*
- *$\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation*

such that:

1. $(l,u) \xrightarrow{d} (l,u+d)$ *if* $\forall\ d' : 0 \le d' \le d \Longrightarrow u+d' \in I(l)$, *and*
2. $(l,u) \xrightarrow{a} (l',u')$ *if* $\exists\ e = (l,a,g,r,l') \in E$ *such that* $u \in g$, $u' = [r \mapsto 0]u$, *and* $u' \in I(l')$,

where for $d \in \mathbb{R}_{\ge 0}$, $u+d$ *maps to each clock* x *in* C *to the value* $u(x)+d$, *and* $[r \mapsto 0]u$ *denotes the clock valuation which maps each clock in* r *to 0 and agrees with* u *over* $C \setminus r$. □

Timed automata can be composed into a *network of timed automata* over a common set of clocks and actions, consisting of n timed automata $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$, $1 \le i \le n$. A location vector is a vector $\bar{l} = (l_1, \ldots, l_n)$. We compose the invariant functions into a common function over location vectors $I(\bar{l}) = \wedge_i I_i(l_i)$. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'. The semantics of a network of timed automata is defined as follows.

Definition 3.3.1.3 (Semantics of a network of TA):
Let $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$ *be a network of* n *timed automata. Let* $\bar{l}_0 = (l_1^0, \ldots, l_n^0)$ *be the initial location vector. The semantics is defined as a transition system* $\langle S, s_0, \rightarrow \rangle$, *where*

- $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^C$ *is the set of states,*
- $s_0 = (\bar{l}_0, u_0)$ *is the initial state, and*
- $\rightarrow \subseteq S \times S$ *is the transition relation defined by:*
  1. $(\bar{l},u) \xrightarrow{d} (\bar{l},u+d)$ *if* $\forall\ d' : 0 \le d' \le d \Longrightarrow u+d' \in I(\bar{l})$.
  2. $(\bar{l},u) \xrightarrow{a} (\bar{l}[l_i'/l_i],u')$ *if* $\exists\ l_i \xrightarrow{\tau g r} l_i'$ *such that* $u \in g, u' = [r \mapsto 0]u$ *and* $u' \in I(\bar{l}[l_i'/l_i])$.
  3. $(\bar{l},u) \xrightarrow{a} (\bar{l}[l_j'/l_j, l_i'/l_i],u')$ *if* $\exists\ l_i \xrightarrow{c?g_i r_i} l_i'$ *and* $l_j \xrightarrow{c!g_j r_j} l_j'$ *such that* $u \in (g_i \wedge g_j), u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

□

3.3.2 Graphical notation in Uppaal

In this thesis timed automata will be used for the definition of the operational semantics for all modeling elements introduced in Chapter 5.

Instead of pure timed automata we will use the Uppaal extended timed automata model. Uppaal [233] is a toolbox for verification of real-time systems. The tool is designed to verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronization. In the following, the graphical notation used by Uppaal will be briefly introduced.

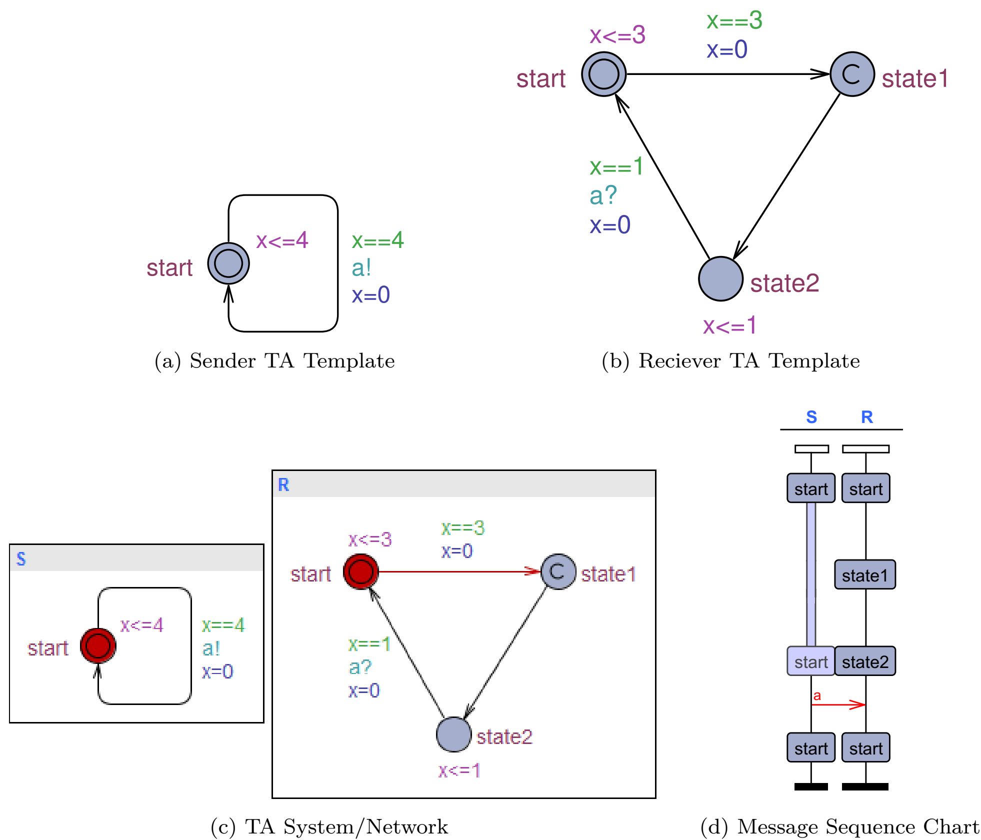
Figure 3.1 gives a graphical example of a timed automata network in Uppaal. The example consists of two *timed automata templates* `Sender(chan &a)` (3.1a) and `Receiver(chan &a)` (3.1b) each of them has a *local clock* `x`.

The sender automata performs a *binary synchronization* on channel `a` every 4 time units. The *invariant* `x<=4` in the *initial location* `start` specifies, that this location can be active for up to 4 time units. The *clock guard* `x==4` on the transition requires to leave location `start` at exactly 4 time units, the synchronization on channel `a` is performed and the clock is set to start counting from zero again (`x=0`).

The receiver automata works in a similar way. The initial location `start` is active for 3 time units. The following location `state1` is a *committed location*. This special location is semantically equivalent to adding an extra clock `y`, that is reset on all incoming edges, and having an invariant `y<=0` on the location. Hence, time is not allowed to pass when the system is in an committed location. Furthermore, a committed location cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations. Location `state2` is active for 1 time unit and synchronizes with channel `a` before returning to the initial location.

Figure 3.1c visualizes the instantiation of the timed automata templates `Sender` and `Receiver`, cp. Listing 3.1.

Figure 3.1d shows the Message Sequence Chart of a symbolic simulation of the timed automata system from Figure 3.1c. Each automata's lifeline (vertical lines) depict the sequence of active locations. The synchronization via channel `a` is shown as a horizontal line.



(a) Sender TA Template (b) Reciever TA Template

(c) TA System/Network (d) Message Sequence Chart

Figure 3.1: Graphical representation of a timed automata network in Uppaal

```
1 chan a;
2 S ← Sender(a);
3 R ← Receiver(a);
4 system S, R;
```

Listing 3.1: Timed automata system shown in Figure 3.1c

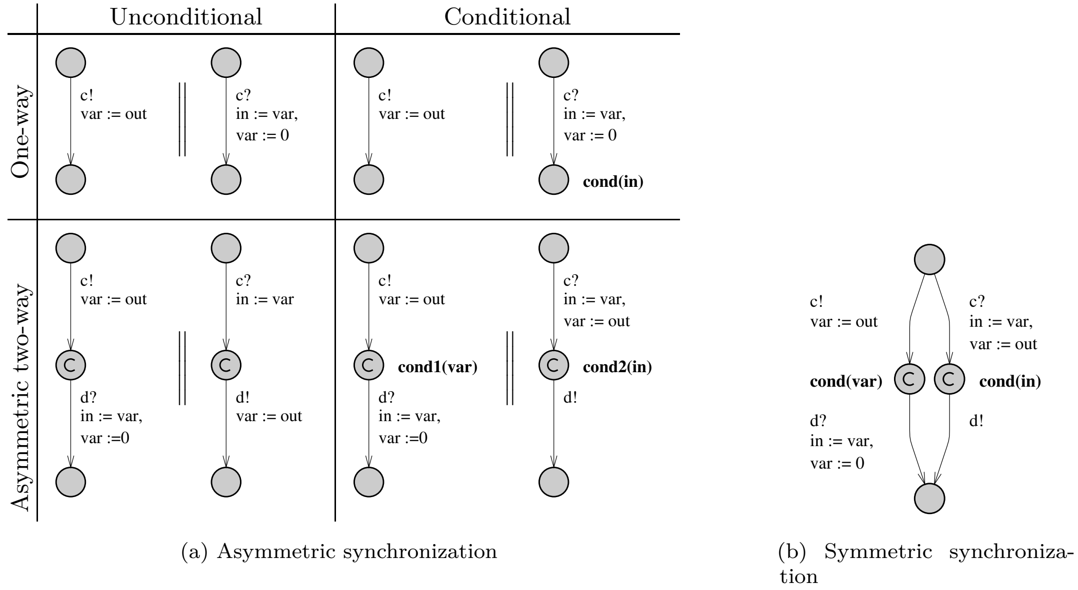
3.3.3 Synchronous Value Passing

The different flavors of sender and receiver synchronization over shared binary channels and data exchange via shared variables[1] can be modeled in Uppaal. More specifically, Uppaal enables modeling of asymmetric and symmetric sender and receiver synchronization. In *asymmetric synchronization* one process acts as sender and another process acts as sender, see Figure 3.2a. In *symmetric synchronization* a process can be sender and receiver at the same time, i.e. a process can non-deterministically choose to act as either the sender or the receiver (see Figure 3.2b).

In [120] the following four variations of *Synchronous Value Passing* are described:

> "In one-way value passing a value is transferred from one process to another, whereas two-way value passing transfers a value in each direction. In unconditional value passing, the receiver does not block the communication, whereas conditional value passing allows the receiver to reject the synchronization based on the data that was passed.
>
> In all four cases, the data is passed via the globally declared shared variable `var` and synchronization is achieved via the global channels `c` and `d`. Each process has local variables in and out. Although communication via channels is always synchronous, we refer to a `c!` as a send-action and `c?` as a receive-action.

[1]Uppaal evaluates the assignment of the sending synchronization first, the sender can assign a value to the shared variable which the receiver can then access directly.



(a) Asymmetric synchronization

(b) Symmetric synchronization

Figure 3.2: Sender and receiver synchronization styles in timed automata [120]

In one-way value passing only a single channel `c` and a shared variable `var` is required. The sender writes the data to the shared variable and performs a send-action. The receiver performs the co-action, thereby synchronizing with the sender. Since the update on the edge with send-action is always evaluated before the update of the edge with the receive-action, the receiver can access the data written by the sender in the same transition. In the conditional case, the receiver can block the synchronization according to some predicate `cond(in)` involving the value passed by the sender[2].

Two-way value passing can be modeled with two one-way value passing pattern with intermediate committed locations. The committed locations enforce that the synchronization is atomic. Notice the use of two channels: In the conditional case each process has a predicate involving the value passed by the other process. The predicates are placed on the invariants of the committed locations and therefore assignment to the shared variable in the second process must be moved to the first edge." [120]

3.3.4 Properties

Uppaal uses a simplified version of TCTL (Timed Computation Tree Logic). Like in TCTL, the query language consists of path formulae and state formulae. State formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be classified into reachability, safety and liveness. Figure 3.3 illustrates the different path formulae supported by Uppaal. Each type is described below.

3.3.4.1 State Formulae

A state formula is an expression that can be evaluated for a state without looking at the behavior of the model. For instance, this could be a simple expression, like `i == 7`, that is true in a state whenever `i` equals 7. It is also possible to test whether a particular process is in a given location using an expression on the form `P.l`, where `P` is a process and `l` is a location.

In Uppaal, deadlock is expressed using a special state formula. The formula simply consists of the keyword `deadlock` and is satisfied for all deadlock states. A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. Due

[2]The intuitive placement of this predicate is on the guard of the receiving edge. Unfortunately, this will not work as expected, since the guards of the edges are evaluated before the updates are executed, i.e., before the receiver has access to the value. The solution is to place the predicate on the invariant of the target location.

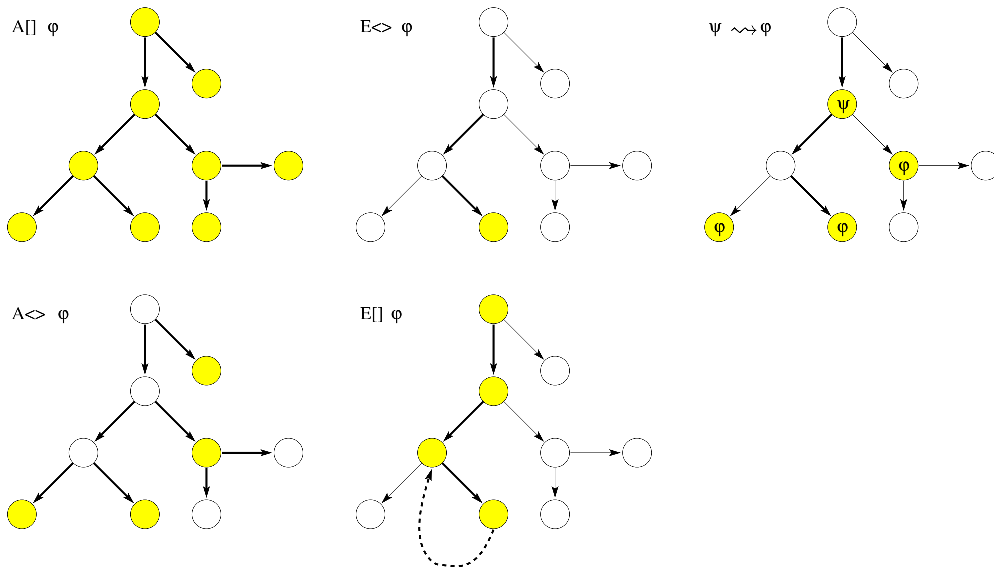


Figure 3.3: Path formulae supported in Uppaal. The filled states are those for which a given state formulae ϕ holds. Bold edges are used to show the paths the formulae evaluate on [120].

to current limitations in Uppaal, the deadlock state formula can only be used with reachability and invariantly path formulae.

3.3.4.2 Reachability Properties

Reachability properties ask whether a given state formula, φ, *possibly* can be satisfied by any reachable state. Another way of stating this is: Does there exist a path starting at the initial state, such that φ is eventually satisfied along that path. The path formula $E\Diamond\varphi$ expresses that some state satisfying φ should be reachable. In Uppaal, this property is expressed by `E<>` φ.

3.3.4.3 Safety Properties

Safety properties are of the form: "something bad will never happen". A variation of this property is that "something will possibly never happen". In Uppaal these properties are formulated positively, e.g., something good is invariantly true. Let φ be a state formulae. The path formulae $A\Box\varphi$ expresses that φ should be true in all reachable states. $E\Box\varphi$ says that there should exist a maximal path[3] such that φ is always true. In Uppaal the properties are expressed by `A[]` φ and `E[]` φ, respectively.

3.3.4.4 Liveness Properties

Liveness properties are of the form: "something will eventually happen". In its simple form, liveness is expressed with the path formula $A\Diamond\varphi$, meaning φ is eventually satisfied. The more useful form is the leads to or response property, written $\varphi \leadsto \psi$ which is read as whenever φ is satisfied, then eventually ψ will be satisfied. $\varphi \leadsto \psi$ is equivalent to $A\Box(\varphi \Rightarrow A\Diamond\psi)$. In Uppaal these properties are written as `A<>` φ and φ `->` ψ, respectively.

For more details about the specific Uppaal extension, refer to the Uppaal tutorial [120].

[3] A maximal path is a path that is either infinite or where the last state has no outgoing transitions.

3.4 Model of Computation, Architecture, Structure and Performance

Definition 3.4.0.1 (Model):
A model is a limited image of the (physical) reality. According to Herbert Stachowiak (General Model Theory [202] page 131-133) a model is characterized by at least three attributes:

1. ***Image*** *- A model is always a model of something, an image representation of a natural or an artificial original, which itself can be a model again.*
2. ***Reduction*** *- in general a model does not capture all attributes of the original, but only those that appear to relevant to the creator of the model.*
3. ***Pragmatism*** *- models are not uniquely associated with their original per se. They fulfill their replacement function*
   a) *for certain subjects (for whom?),*
   b) *within certain time intervals (when?), and*
   c) *under certain restrictions on mental or physical operations (for what?).*

□

In this work models will be used to obtain different views of a System on Chip (SoC) under design. The main purpose of our models is to observe, predict and reason about properties of a SoC design. In particular we will distinguish between the following models:

3.4.1 Model of Computation (MoC)

We use the term *Model of Computation* (MoC) to focus on issues of concurrency and time. In literature different definitions can be found [99, 170, 127, 173, 174]. In this work a MoC defines the time representation and the semantics of communication and synchronization between processes in a process network. Thus, a MoC defines how computation takes place in a structure of concurrent processes, hence giving a semantics to such a structure [134, 122]. This semantics can be used to formulate an abstract machine that is able to execute a model.

As stated in [99] a model of computation should support the following properties:

Implementation independence An abstract model should not expose details of a possible implementation, e.g. which kind of processor used, how much parallel resources available, what kind of hardware implementation technology used, details of the memory architecture, etc. Since a model of computation is a machine abstraction, it should by definition avoid unnecessary machine details. The benefits of an abstract model include, that analysis and processing is faster and more efficient, that analysis results are relevant for a larger set of implementations, and that the same abstract model can be directed to different architectures and implementations. The drawback of this abstraction is is reduced analysis accuracy due to a lack of knowledge of the target architecture that can be exploited for modeling and design.

Composability Since many parts and components are typically developed independently and integrated into a system, it is important to avoid unexpected interferences. Thus some kind of composability property [136] is desirable. One step in this direction is to have a deterministic computational model such as Kahn process networks, that guarantees a particular behavior independent of the time individual activities and independent of the amount of available resources in general.

Analyzability A general trade-off exists between the expressiveness of a model and its analyzability. By restricting models in clever ways, one can apply powerful and efficient analysis and synthesis methods. For instance, the Synchronous Data Flow (SDF) model [195] allows all actors only a constant amount of input and output tokens in each activation

cycle. While this restricts the expressiveness of the model, it allows to efficiently compute static schedules when they exist. For general datafow graphs this may not be possible because it could be impossible to ensure that the amount of input and output is always constant for all actors, even if they are in a particular case. Since SDF covers a fairly large and important application domain, it has become a very useful model of computation.

To choose the "right" model of computation is of utmost importance, since each MoC has certain properties, regarding expressiveness and analyzability. Generally speaking there is a trade-off between expressiveness and analyzability: The more freedom and flexibility a MoC allows to the designer, the less properties (like schedulability, deadlock freedom, or memory boundedness) can be guaranteed. Depending on the application (data flow or control dominated) and the requirements on the overall system (e.g. hard real-time system, no real-time constraints) the MoC is chosen.

Following [122, 99] MoCs can be organized according to their time abstraction. In this work the following MoCs will be considered:

- **untimed sequential:** Pure functional models, written in the sequential programming languages C and C++ will be used to describe functionality, see Section 3.6.

- **untimed process networks:** For explicitly expressing concurrency or explicitly specifying potential concurrency at application level, untimed sequential blocks can be composed in an untimed process network. Classical examples for untimed process networks are Kahn Process Networks (KPN) [200] and Synchronous Data Flow (SDF). In this work we consider a more flexible MoC, called Program State Machine (PSM) (see Section 3.6.2). This model is a superset of KPN and SDF and allows the integration of state-based MoCs, like Finite State Machines (FSM) and their combination with data flow oriented models. While PSMs can be used to capture untimed process networks, the leave behavior blocks of PSMs can also be annotated with execution times, leading to a timed MoC.

- **discrete event:** A PSM MoC with annotated execution times is a timed discrete event model. For the implementation of an executable timed PSM model we will use SystemC (see Chapter 6) that implements discrete event semantics.

- **continuous time:** To formally define and analyze the protocol, synchronization behavior and timing properties of a timed process network we will make use of Timed Automata (TA) (see Section 3.3). Timed Automata are using a continuous time model.

Figure 3.4 gives an overview of the different classes of Models of Computation used in this work. Each point (A) - (F) represents a different design model. The axes span up a computation and communication refinement space from un-timed specification over approximate-timed intermediate to cycle-timed implementation models. The reason for separating computation and communication refinement is to support the "divide and conquer" principle supported by modern system-level design languages (ref. "orthogonality" in Section 3.6.1). The arrows between the design points describe possible refinement paths. The red solid arrows depicts the shortest path from a specification to an implementation model.

In this thesis the following combinations of Models of Computation and time models will be used for the different design models:

- **(A):** The Program State Machine (PSM) MoC (see Section 3.6.2) is used for un-timed specification models. The implementation of this MoC is realized through mapping it to a Discrete Event (DE) time model (see Section 6.3).

- **(B),(C):** Approximate-timed computation and un-timed (B) and approximate-timed (C) communication models will be represented through Timed Communicating Sequential Process (TCSP) (see Section 5.5). In this model the execution time of computations and the communication time is annotated in dense time intervals (i.e. continuous time) using a timed automata representation (see Chapter 5). The implementation of the TCSP MoC is realized through mapping it to a Discrete Event (DE) time model (see Section 6.4).

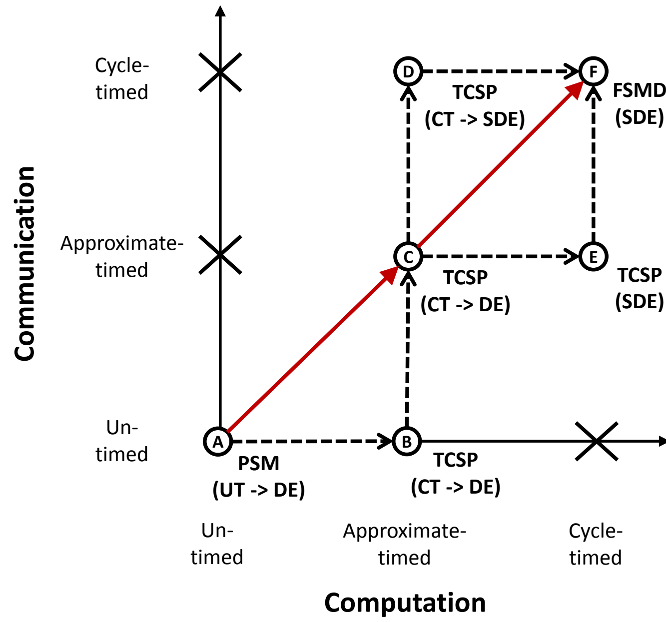


Figure 3.4: Classes of Models of Computation used in this work (PSM = Program State Machine, TCSP = Timed Communicating Sequential Process, FSMD = Finite State Machine with Data-path) and their time models (UT = Un-Timed, DE = Discrete Event, SDE = Synchronous Discrete Event, CT = Continuous Time). The "->" symbol denotes a mapping from the specification to the implementation time domain (chart inspired by [131]).

- **(D):** Same as (B) and (C). Due to the clock-cycle accuracy of communication, the implementation of the TCSP MoC is realized through mapping it to a Synchronous Discrete Event (DE) time model (see Section 6.5). The overall system description is synchronous to a global clock signal.
- **(E):** Same as (D), but without a continuous time representation in the specification model.
- **(F):** The final implementation system is represented as a network of clock cycle accurate Finite State Machine with Datapath (FSMD) realized in a Synchronous Discrete Event model.

In this work we will focus on: (A) *Behavioral Layer Model* (see Section 6.3), (B) *Application Layer Model* (see Section 6.4), (C)(D) *Virtual Target Architecture Layer Model* (see Section 6.5), and (F) *Implementation Model* (see Section 7.4).

3.4.2 Model of Architecture (MoA)

A *Model of Architecture* describes an architecture template, e.g., available resources (processing elements, storage elements, and communication elements), their capabilities (or services) and their interconnections. As well as the classification of behavioral models into MoCs, specific ways of describing architecture templates can be generalized into MoAs [143]. In this sense a MoA describes the characteristics underlying a class of platform models in order to evaluate richness of supported target architectures. Such architecture templates can be coarsely subdivided based on their processing, memory and communication hierarchy [55]. For *processing elements*, this includes single-processor systems, hardware/software processor or co-processor systems, homogeneous (symmetric), and heterogeneous (asymmetric) multi-processor/multi-core systems (MPSoCs). *Memory elements* can be subdivided into shared and distributed memory architectures. *Communication architectures* can be grouped into shared bus-based, dedicated point-to-point or packet-based Network-on-Chip (NoC) approaches. Beside the architecture template, constraints typically contain mapping restrictions and additional constraints on extra-functional properties like maximum and minimum clock frequency per architecture element,

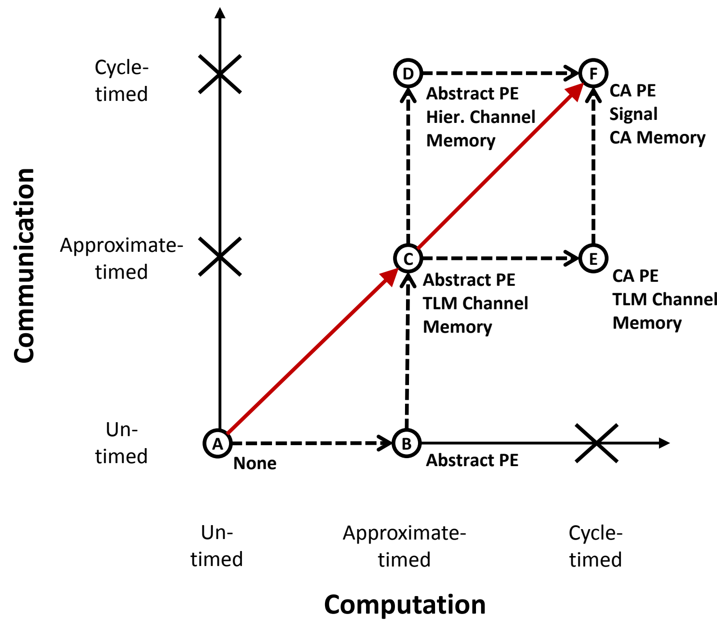


Figure 3.5: Classes of Models of Architecture used in this work, PE = Processing Element, CA = Cycle-Accurate (chart inspired by [131])

minimum/maximum size of memory elements, and minimum/maximum width of communication elements.

In this work we will consider a reduced set of MoAs with the following elements:

Processing Elements can be single processors with local data and instruction memory including communication interface to a bus or dedicated point-to-point communication infrastructure. We call this processing element with local memory *Processing Element Tile*. Different of these Processing Element Tiles can can be combined to a heterogeneous multi-processor system. Each Tile of this heterogeneous system can have a different kind of single processor (i.e. micro- and instruction-set architecture) including peripheral components.

Memory Elements are distributed among Processing Element Tiles to facilitate communication between applications running on different Tiles. This distributed and shared memory is protected for concurrent access using special communication wrappers, called *Shared Object*.

Communication Elements are shared buses with configurable protocol and bus width, and dedicated bidirectional point-to-point channels with configurable send and return bitwidth.

Figure 3.5 provides an overview of classes of Models of Architecture used in this work. The semantics of the diagram are the same as for Figure 3.4.

In this thesis the following Model of Architecture elements will be used:

- **(A):** No architectural elements are used.
- **(B):** Abstract Processing Elements (PEs) are used to describe the entity of a processor (which can be a software processor or custom hardware block). Each PE is an abstraction from its synthesizable and cycle accurate implementation and only represents a) a computing resource as a structural block, b) interfaces to communication channels c) extra-functional properties or requirements (e.g. min and max clock frequency, technology node dependent chip area).
- **(C):** Same as in model (B) but with communication between PEs and Memories are specified by TLM Channels. A TLM channel abstracts from the cycle accurate protocol and the internal structure of an on-chip bus or point-to-point connection. Communication is

described through send and receive service calls with annotated delays. Channel arbitration is can be modeled on transaction granularity. Memory elements represent different kinds of on- and off-chip memories including their controllers. Memory blocks accurately capture the total size and read/write delays of their physical counterparts. The memory layout (padding and alignment) is not captured, assuming a perfectly packed memory.

- **(D):** Same Abstract Processing and Memory Elements as used in (C). Communication between PEs and Memories are specified by a bit and cycle accurate bus protocol description. For connecting this cycle accurate communication model with the approximately (non-bit accurate) PE and Memory models transactor components are used. These transactors translate from send and receive service calls to the bit and cycle accurate channel protocol. The transactors and the bit and cycle accurate channel models are combined in a Hierarchical Channel model (see Section 6.5.3).
- **(E):** Same communication channel and memory elements as used in (C). All PEs are described on a bit and cycle-accurate level. For software processors this is an Instruction Set Architecture (ISA) accurate specification. For custom hardware it is a clock cycle and bit accurate FSMD specification.
- **(F):** Same processing element granularity as used in (E) and same communication channel granularity as used in (D), but without transactors because all connected components are specified at bit and cycle accurate granularity. The Memory Elements are specified bit and cycle-accurate including representation of processing element specific memory layout (alignment and padding).

3.4.3 Model of Structure (MoS)

Based on the combination of MoCs for component-internal behavior and functional semantics, the term *Model of Structure* for separate classification of such implementation representations and their architectural or structural semantics can be introduced. A MoS allows characterization of the underlying abstracted semantics of a class of structural models independent of their syntax. Hence, MoSs can be used to compare expressibility and analyzability of specific implementation representations. For example, at many levels a netlist concept is used with semantics limited to describing component connectivity. At the system level, pin-accurate models (PAMs) combine a netlist with bus-functional component models. Furthermore, transaction-level modeling (TLM) concepts and techniques are employed to abstract away from pins and wires. Similar to behavioral models, structural models are often represented in a programming language, system-level description language (SLDL) or hardware description language (HDL).

In this work the following different MoSs will be used to represent the structure of the system at different abstraction levels (see Figure 3.6):

- **Hierarchical Behavior Model** for capturing a pure sequential functional system description in a hierarchical parallel implementation independent structure.
- **Flat Process Network Model** represents a computational refined pre-implementation structure of the application. Processes contain sequentially scheduled behaviors and communicate through Shared Objects. A process can be implemented in software or in hardware. Shared Objects are implemented as dedicated hardware resources.
- **Transaction Level Model** represents an implementation model with processes mapped to Processing Element Tiles or dedicated Hardware Processing Elements. Shared Objects are represented as dedicated Hardware Communication Processing Elements. Communication between Processing Element Tiles, dedicated Hardware Processing Elements, and Hardware Communication Elements is represented by Routing Elements that represent configurable shared buses of point-to-point channels. Each Routing Element implements a transport function that transfers a Communication Payload (Transaction) from a sender to its receiver. These routers abstract from any signal level structural details.
- **Pin Accurate Netlist Model** represents a Transaction Level Model with signal level structural details of each Communication/Routing Element.

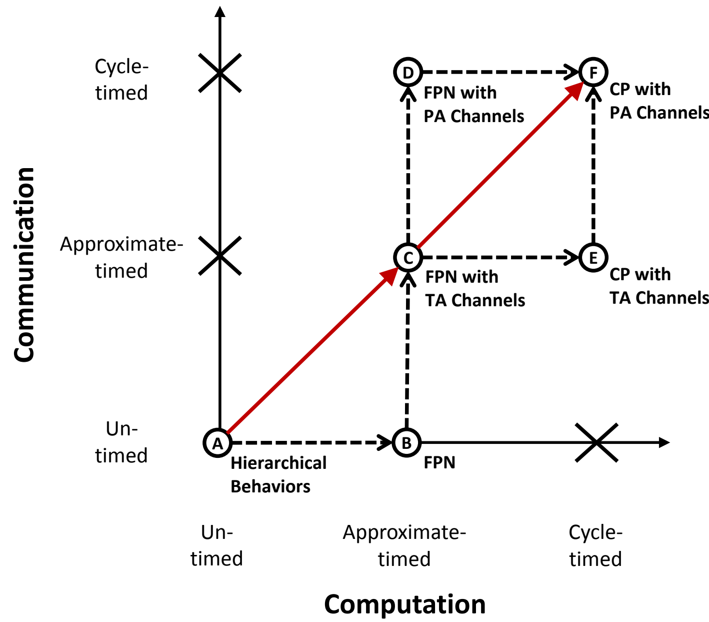


Figure 3.6: Classes of Models of Structure used in this work, FPN = Flat Process Network, TA = Transaction-Accurate, PA = Pin-Accurate, CP = Clocked Process (chart inspired by [131])

3.4.4 Model of Performance (MoP)

A *Model of Performance* refers to the overall accuracy and granularity in time and memory space. Generalizing from the detailed definitions of specific performance models, such as timing, power or cost/area models, a MoP can be used to judge the accuracy of the quality numbers and the computational effort to get them. Quality numbers are often used as objective values during design space exploration when identifying the set of optimal or near-optimal implementations.

Figure 3.7 gives a graphical representation of different classes of timing granularity for Models of Performance. The two axes span a vector space with independent *Computation* and *Communication* components. This construction enables different timing granularity for computation and communication during the top-down refinement. In Section 3.6.2 the rational behind this separation of computation and communication will be described in more detail. The arrows between the different MoPs define possible paths in the refinement from an untimed system specification (A) to a full cycle-accurate model (F).

In this thesis the following timing granularities will be considered:

(A) **Untimed Functional Model (UFM)** is a pure functional model with a defined execution order (this can be a partial order to express concurrency) of functions and explicit synchronization between partly ordered functions. UFMs are used to capture causality in system specification models.

(B) **Timed Functional Model (TFM)** is an extension of the UFM with approximated execution time/duration annotations for computation elements.

(C) **Transaction-Level Model (TLM)** is an extension of the UFM with execution time/duration annotations for computation elements and approximated communication time/duration annotations of communication elements.

(D) **Bus Cycle-Accurate Model (BCAM)** is an extension of the TLM with execution time/duration annotations for computation elements and cycle accurate communication elements.

(E) **Computation Cycle-Accurate Model (CCAM)** is an extension of the TLM with cycle accurate computation elements and approximated communication time/duration annotations of communication elements.

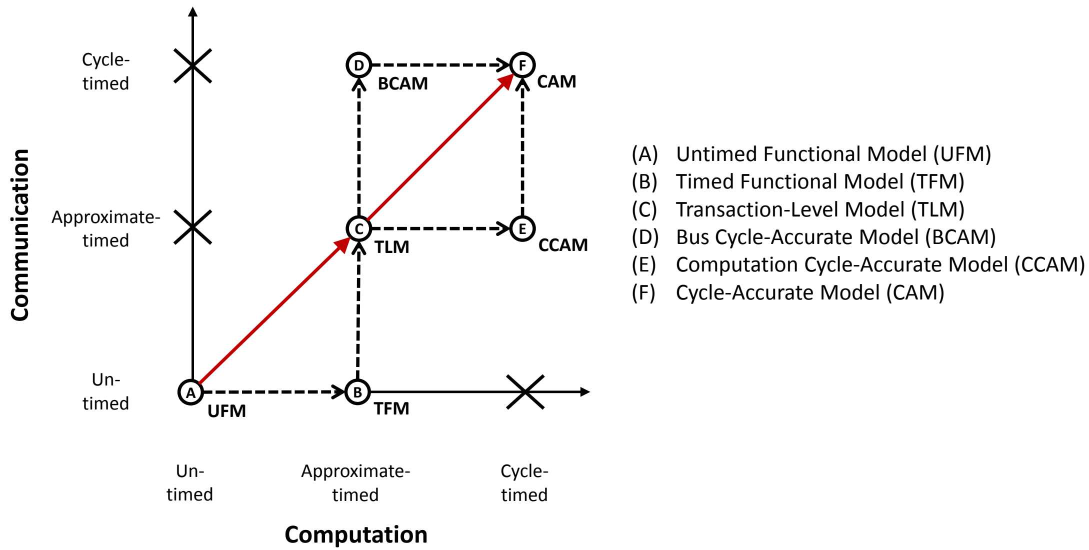


Figure 3.7: Classes of timing granularity for Models of Performance (based on [27])

(F) **Cycle-Accurate Model (CAM)** is an extension of the TLM with cycle accurate computation and communication elements.

In this work we will focus on (A) Untimed Functional Models (UFM) for specification models, (B) Timed Functional Models (TFM) for times specification models, (C) Transaction Level Models (TLM) for hardware/software partitioned models, (D) Bus Cycle-Accurate Models (BCAM) for the virtual platform models, and (F) Cycle-Accurate Models (CAM) for the implementation models.

Performance models can be subdivided into the following three classes:

Simulative performance models require an executable representation of the Design Under Test (DUT), an executable input model that generates stimuli for the DUT, and an executable monitor that compares the results of the DUT for each stimulus with the expected result. Simulation techniques are widely used for functional test and performance analysis. The quality and value of a simulation depends on the chosen set of test vectors in the stimuli set. The main advantage of simulative performance models is that they are able to capture the function and structure of very complex systems on different abstraction levels. The main drawback of simulative approaches is their incompleteness regarding the simulation of all possible combination of input values.

In this work we will use SystemC [219] to implement a simulative performance model at the timing granularities: (A) Untimed Functional Models (UFM), (B) Timed Functional Models (TFM), (C) Transaction Level Models (TLM) and (D) Bus Cycle-Accurate Models (BCAM). VHDL [236] and Verilog [237] will be used as simulative performance model for (F) Cycle-Accurate Models (CAM).

Analytical performance models are based on an abstract model of the implementation model including (over-)approximated properties, like Best-Case Execution Time (BCET), Worst-Case Execution Time (WCET), Best-, and Worst-Case communication delay, etc. These techniques have the advantage over the simulation-based approaches that system properties can be calculated independent from input data stimuli. While simulation-based analysis gives performance evaluation for a specific input experiment, analysis-based approaches evaluates the system for all possible input stimuli, thus giving a safe upper-bound on specific performance measures. The drawback of analytical models is that their results might be too pessimistic or that they suffer from complexity problems resulting in huge analysis times (sometimes inappropriate for real applications).

In this work Timed Automata [191] in UPPAAL [233, 120, 175] are used to formally express the execution semantics of the system at the timing granularities (B) Timed Functional Models (TFM) and (C) Transaction Level Models (TLM). The corresponding Timed Automata representation can be used to check reachability, safety and liveliness properties which cannot be checked with a pure simulative approach. Analysis of these properties using model-checking is not part of this thesis. For an application of timing analysis of SDF applications mapped on shared bus architectures in TLMs using model-checking refer to [4].

Hybrid approaches try to offer the best properties of simulative and analytical MoPs. Analytical methods are used to generate stimuli data for a simulation model. These generated stimuli data can originate from an abstract implementation model with (over-)approximated extra-functional properties. Thus simulation runs of the implementation system with realistic and accurate extra-functional properties are used to check whether reported property violations are really met. From an analytical model stimuli data for simulations can be generated to serve different purposes:

a) functional coverage based on function point analysis
b) timing analysis based on (critical) path analysis
c) corner-case analysis for extra-functional properties

Hybrid approaches are not covered in this thesis. It is envisioned that future work based on [4] can be combined with the simulative approach presented in this thesis.

3.4.5 Summary

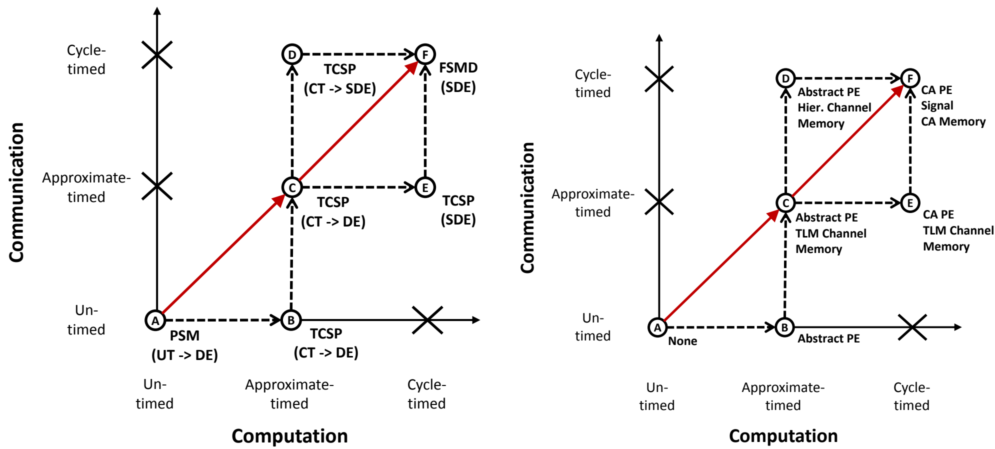
To summarize the different presented models and to link them together we are using the X-Chart (see Figure 3.8c) as introduced in [28]. The X-Chart describes a synthesis process that takes as inputs a specification model, consisting of a *Behavior* description and *Constraints*, and explicit designer decisions. The output of this synthesis process is an implementation model, where the behavior specification gained *structural properties* and *quality numbers* enable the assessment of the refinement decisions.

The behavioral model represents the intended functionality of the system. Its expressibility and analyzability can be declared by its underlying Model of Computation (MoC). The constraints often include an implicit or explicit platform model that describes an architecture template, e.g., available resources, their capabilities (or services) and their interconnections. Analogous to the classification of behavioral models into MoCs, specific ways of describing architecture templates can be generalized into Models of Architecture (MoA).

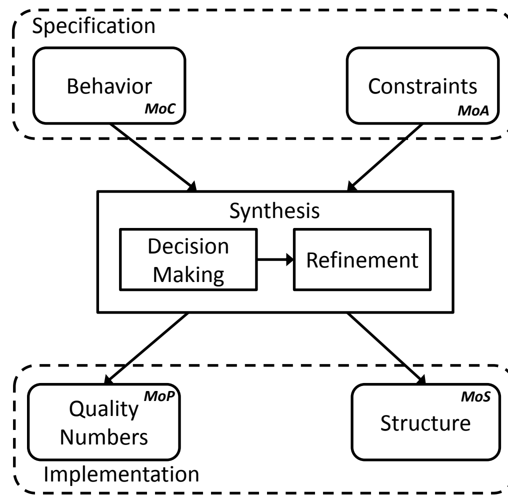
The structural model is a refined model from the behavioral model under the constraints of the specification. In addition to the implementation-independent information of the behavioral model, the structural model contains information about the realization of design decisions from the previous synthesis step, i.e., mapping of the behavioral model onto an architecture template. A structural model is a representation of the resulting architecture as a composition of components that are internally described as behavioral models for input to the next synthesis step. The underlying Model of Structure (MoS) can be used for separate classification of such implementation representations and their architectural or structural semantics. The quality numbers, like throughput, latency, response time and area, can be described by the concept of an underlying Model of Performance (MoP). As described above, a MoP refers to the overall accuracy and granularity in time and space.

Using the X-Chart the different MoCs from Figure 3.8a, MoAs from Figure 3.8b, MoPs from Figure 3.8d and MoSs from Figure 3.8e can be combined in the following ways:

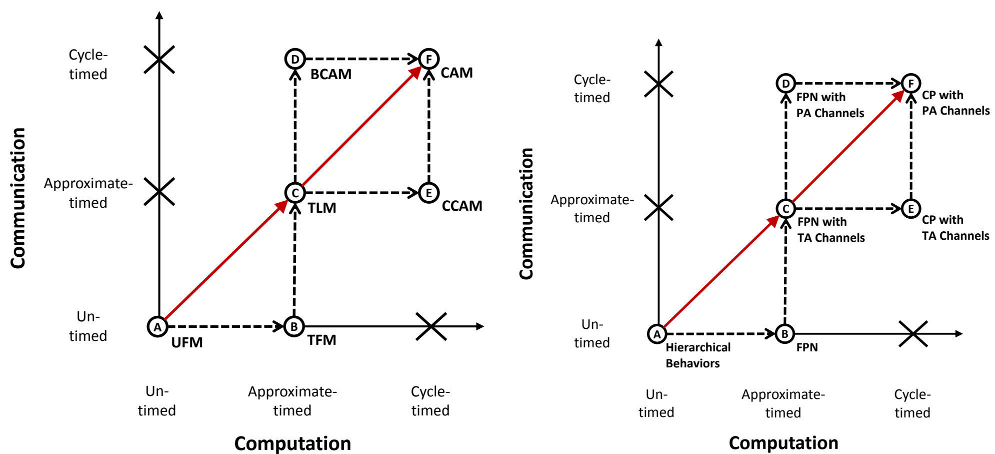
Different views on a single model This static view on a single design model (A)-(F) enables different views on its underlying MoC, MoA, MoP and MoS. E.g. for un-timed computation and communication models (all (A) points) the MoC is defined as a Program State Machine (PSM), it has no associated MoA, is an Un-timed Functional Model (UFM) and its MoS is a hierarchical set of Behaviors. For the cycle-timed computation and communication models



(a) Model of Computation (MoC) (based on [131]) (b) Model of Architecture (MoA) (based on [131])



(c) X-Chart (based on [28])



(d) Model of Performance (MoP) (based on [131]) (e) Model of Structure (MoS) (based on [131])

Figure 3.8: Summary of presented models

(all (F) points) the MoC is defined as a Finite State Machine with Datapath (FSMD), its associated MoA consist of Cycle-Accurate Processing Elements and Memories connected through Signals, is a cycle-accurate MoP and its MoS is a set of Clocked Processes with Pin-Accurate Channels.

Synthesis and refinement of models This view describes the transformation between models through synthesis as defined by the X-Chart and denoted through the arrows between the design models (A)-(F). E.g. to describe the synthesis step from design model (A) to (C) the following mapping to the X-Chart blocks are applied: The *Behavior* is described as a PSM (see Figure 3.8a (A)) the *Constraints* are given as number and types of Abstract PEs, TLM Channels and Memories including their possible interconnections (see Figure 3.8b (C)). The *Quality Numbers* are at TLM granularity (see Figure 3.8d (C)) and the *Structure* is a Flat Process Network with Transaction-Accurate Channels (see Figure 3.8e (C)), whose underlying MoC is a network of Timed Communicating Sequential Processes (TCSP) (see Figure 3.8a (C)). For the synthesis step from model (C) to (F) the implementation model of the previous step is used as the Behavior part of the Specification Model and constraints are described by MoA (F) from Figure 3.8b.

For the proposed methodology Figure 3.9 gives an overview of the mapping of different design points to the modeling layers to be introduced in Chapter 5. The refinement and synthesis steps described and supported by the presented work are indicated by red solid arrows:

(A) → **(B)** describes the refinement of an un-timed specification model of the *Behavioral Layer* to a computation approximate- and communication un-timed model of the *Application Layer*.

(B) → **(D)** describes the refinement of an Application Layer model to an approximate- computation and cycle-timed communication model representative of the *Virtual Target Architecture Layer*. This step will also be called *Application to Virtual Target Architecture Layer mapping*.

(D) → **(F)** describes the synthesis of an approximate- computation and cycle-timed communication model representative of the *Virtual Target Architecture Layer* to a cycle-times computation and communication model of the *Implementation Layer*. This step will also be called *Synthesis* (see Chapter 7).

In the following section a more general overview and taxonomy of *Design Methodologies* will be given.

3.5 Methodology

Definition 3.5.0.1 (Design Methodology):
A Methodology is generally a guideline system for solving a problem, with specific components such as phases, tasks, methods, techniques and tools. A methodology can be considered to include multiple methods, each as applied to various facets of the whole scope of the methodology.

The Y-Chart [197, 193, 143] (see Figure 3.10) is a framework to reason about methodologies. Models are points in the chart at particular level and view. MoCs are classification of behavioral concepts, MoS and MoA for structural concepts. The Y-Chart makes the assumption that each design, can be modeled in three basic ways, which emphasize different properties of the same design. For this purpose the Y-Chart defines

- *three orthogonal views: behavior (sometimes called functionality or specification), design structure (also called netlist or a block diagram), and physical design (sometimes called layout or board design) defining geometrical aspects,*
- *four abstraction levels: circuit, logic, processor, and system. Where all three views can be found on each abstraction level.*

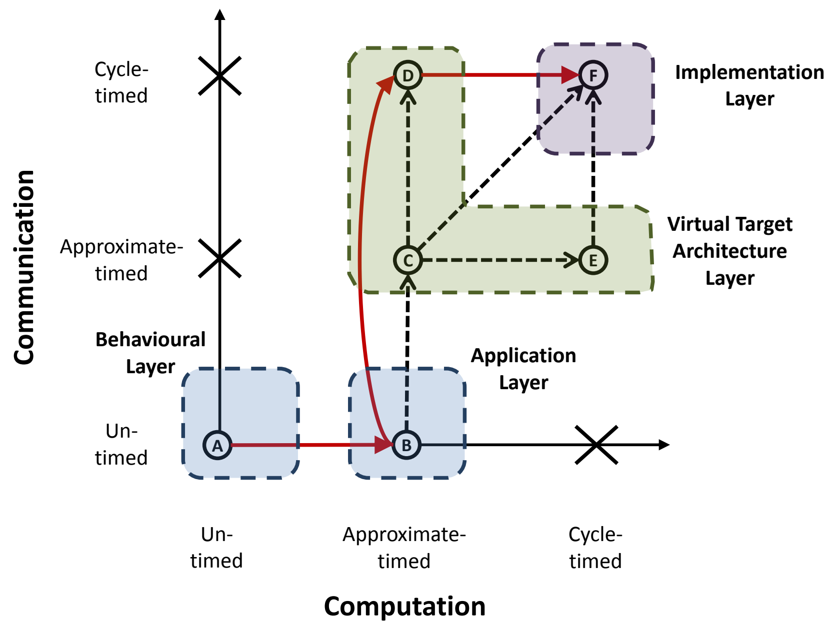


Figure 3.9: Overview of modeling layers for the proposed methodology. Red arrows indicate refinement/synthesis steps supported by the methodology. Dotted arrows indicate refinement/synthesis steps not described in this work.

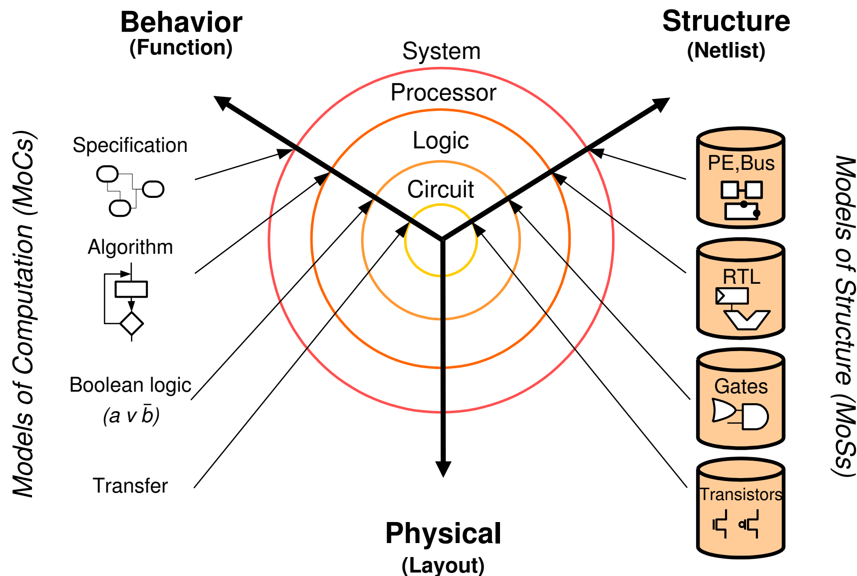


Figure 3.10: Y-Chart [27]

Transformation of a design from one view to another is called synthesis. With regard to the synthesis and refinement steps in the Y-Chart, design methodologies can be classified, e.g. into Top-Down, Bottom-Up, and Meet-in-the-middle (more information can be found in [27], Section 2.1 - 2.6). □

The *Behavior View* represents a design as a black box and describes its outputs in terms of its inputs over time. The black-box behavior does not indicate in any way how to build the black box or what its internal structure is. Anyhow, the interconnection and synchronization and communication semantics of connected black-boxes is specified by an associated MoC. It is also possible that different black-boxes have different associated MoCs. In this case, for communication across MoC boundaries special converter elements need to be provided.

In the *Structural View* the black-box is represented as a set of components and connections. Naturally, the behavior of the black box can be derived from its component behaviors and their connectivity. However, such a derived behavior may be difficult to understand since it is "obscured" by the details of each component and connection.

The *Physical* or *Geometrical View* adds dimensionality to the structure. It specifies the size (height and width) of each component, the position of each component, as well as each port and connection on the silicon chip, printed circuit board, or any other container.

The Y-Chart can also represent a design on different abstraction levels, which are identified by concentric circles around the origin. Typically, four levels are used: circuit, logic, processor, and system.

Components on the circuit level are standard cells which consist of N-type or P-type transistors. On the logic level logic gates and flip-flops are used to generate register-transfer components. These are represented by storage components such as registers and register files and by functional units such as ALUs and multipliers. On the processor level, standard and custom processors, or special-hardware components such as memory controllers, arbiters, bridges, routers, and various interface components are used. And finally on the system level, systems consisting of processors, memories, buses, and other processor components.

A database of components to be used in building the structure for a given behavior is used on each abstraction level. The process of converting the given behavior into a structure on each abstraction level is called synthesis. After definition and verification of the structural representation of the design, the next lower abstraction level is reached by further synthesizing each of the components in the structural view. If each component in the database is given with its structure and physical dimensions, the physical design can be started which consists of floorplanning, placement, and routing on the chip, 3 D stack or printed circuit board. Thus each component in the database may have up to three different models representing three different axes in the Y-Chart: behavior or function; structure, which contains the components from the lower level of abstraction; and the physical layout of its structure.

3.5.1 Design flow

Definition 3.5.1.1 (Design flow):
A Design Flow is the explicit combination of electronic design automation (EDA) tools to accomplish the design of an integrated circuit. The flow of transformations between models in a design flow is defined by its Design Methodology. Models are core of the design flow definition. The following different activities are applied to models in each step of a design flow:

- ***analysis*** *is the extraction of model properties (e.g. syntax check, structural check, functional check, arithmetic operation count, ...)*
- ***synthesis*** *is the transformation of a behavioral to a structural, and from a structural to a physical model*
- ***verification*** *is an experimental validation or formal proof that the model behavior before synthesis equals the model behavior after synthesis*

The design methodology and modeling flow can be described as a set of models and transformations between models, resulting in a sequence of design models. Along with the design flow a component database on each abstraction level of the associated methodology, and the used analysis, synthesis, and verification tools are defined. □

3.5.2 Simulation

Definition 3.5.2.1 (Simulation):
Simulation is the imitation of the operation of a real-world process or system over time [148]. The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviors of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time. □

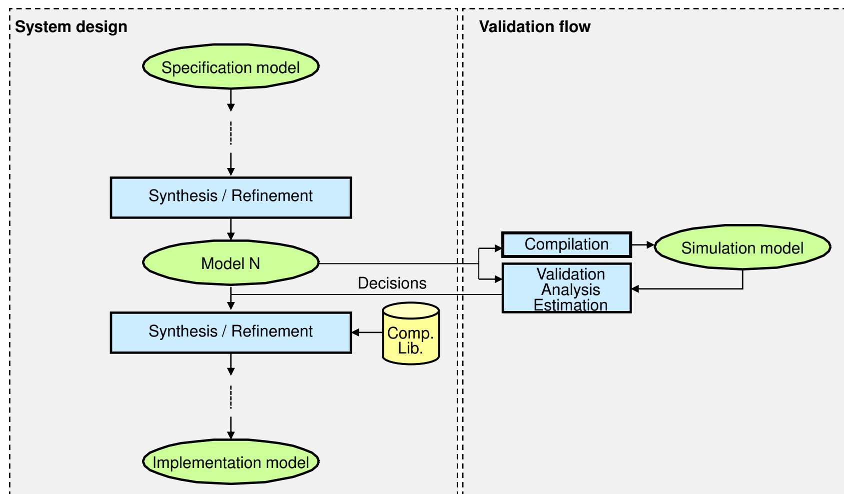


Figure 3.11: Phases of a generic design flow (based on [27])

Alan Turing used the term "simulation" to refer to what happens when a universal machine executes a state transition table (in modern terminology, a computer runs a program) that describes the state transitions, inputs and outputs of a subject discrete-state machine. The computer simulates the subject machine. Accordingly, in theoretical computer science the term simulation is a relation between state transition systems, useful in the study of operational semantics.

In the context of this work simulation is used for the analysis and validation of behavioral and timing properties of different models in the design flow (see Section 5.2). For this purpose the proposed design language has a well-defined operational semantics (see Section 5.5 and Section 6.5) implemented in a discrete event simulation model (see Chapter 6).

3.5.3 Synthesis

Definition 3.5.3.1 (Synthesis):
In general, Synthesis refers to a combination of two or more entities that together form something new; alternately, it refers to the creating of something by artificial means. In the context of this work synthesis is defined as a transition from the behavioral view to the structural and physical view of a model, as represented in the Y-Chart. □

In comparison with the Y-Chart, the X-Chart (as introduced in Figure 3.8) provides a more detailed view on the different input and output models of each possible synthesis step in the Y-Chart. In the Y-Chart synthesis is considered as a repeatable refinement process from the behavior to the structural, and from the structural to the physical view. Synthesis is defined along each of the concentric abstraction level circles of the Y-Chart. The application of the X-Chart scheme is performed either once or several times (depending on the available amount of tool support) on each of these concentric transitions in the Y-Chart.

In this thesis *System Synthesis* and *Processor Synthesis* for embedded SoCs will be covered. An overview of the general meaning and different steps of system and processor synthesis can be found in [27] (Section 1.2.4 and 1.2.7).

3.5.4 Summary

The section gave a definition of the term *Design Methodology*, *Simulation* and *Synthesis* as used in this work. Based on the Y-Chart coordinate system, different design methodologies have been presented. Figure 3.12 provides an outlook on the design methodology used in this work. It is a combination of the system-level (see [27], Section 2.6) and the FPGA-based (see [27], Section 2.5) design methodologies. For building a link to the presented X-Chart approach from

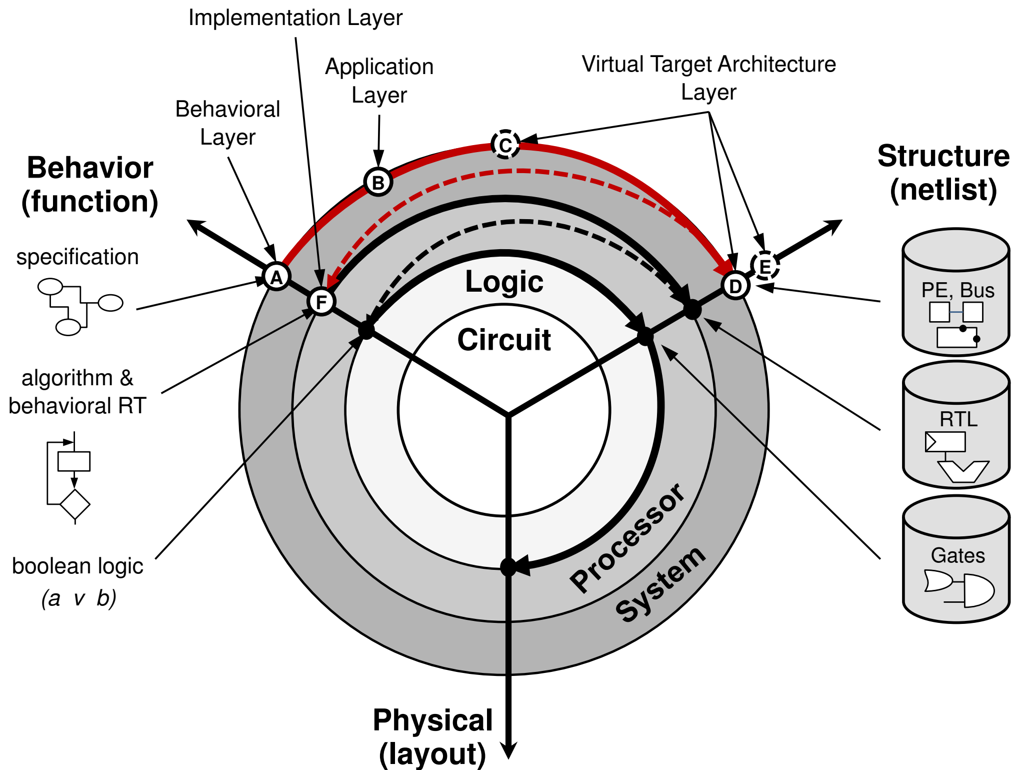


Figure 3.12: Combination of System-level and FPGA-based design methodology as used in this work. The design points (A) - (F) correspond to the X-Chart points in Figure 3.9

Section 3.4, the design points (A) - (F) from Figure 3.9 have been included in the Y-Chart. In Chapter 5 the design point models (A), (B), (D) and (F), and the associated design flow will be presented in more detail.

3.6 System Level Design Representation

This section gives a basic introduction to *System Level Design Languages*. It starts with a generic definition of a design model representation in a *Language*. In our case system level design languages shall be capable of representing an embedded SoC's MoC, MoA, MoS and MoP on different level of abstraction. Furthermore, to enable the (semi-) automatic refinement process between these models in a design flow certain requirements on system level design languages will be formulated.

The *Program State Machine* (PSM) hybrid model (capable to represent different MoCs, MoAs, MoSs and MoPs) will be introduced, since it is the foundation to capture un-timed behavior for the proposed design methodology. Moreover, the flexible channel concept of PSMs is the basic infrastructure to enable the communication refinement and synthesis approach in this work.

Behavior in this work will be captured in an object-oriented model, which is a subset of C++ to enable synthesis for embedded software, communication interfaces (that are both hardware and software) and custom hardware. A brief introduction to the object model and its graph representation will be given.

3.6.1 Language

Definition 3.6.1.1 (Language):
A language represents a model in a machine-readable form. All languages have some primitive building blocks for the description of data and the processes or transformations applied to

them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively. □

The **syntax** of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

Semantics define the meaning of expressions of a language. They does so by evaluating the meaning of syntactically legal strings (symbols) defined by a specific language, showing the computation involved. Semantics describe the processes a machine follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will execute on a certain machine.

A variety of different languages can capture one model. E.g., a sequential program model can be captures in C, C++ or Java. At the same time one language can capture a variety of different models. E.g., C++ is capable to capture a sequential program model, an object-oriented model, a data-flow model or a state machine model. Of course, certain languages are better suited for capturing certain models.

With regard to the different models, introduced in Section 3.4, a System-Level Design Language (SLDL) shall be capable to capture different MoCs, MoAs, MoSs, and MoPs for the design and analysis of embedded SoCs. As stated in [160], SLDLs shall have the following properties:

Executability enables the validation of the represented model through simulation. This property requires the definition of an operational semantics and a simulator that executes the model as defined by the operational semantics of its language. For the simulation of embedded hardware/software systems a discrete event simulation [190, 148] offers a flexible and efficient fundamental technology.

Synthesizability enables the automatic model to model transformation in a design flow. To enable synthesizabiliy the denotational semantics needs to be unambiguous. Each language element from the input model needs to be transformed into one or multiple language elements of the output model without changing operational semantics between input and output model.

Modularity is the degree to which a system's components may be separated and recombined. Systems are deemed "modular", for example, when they can be decomposed into a number of components that may be mixed and matched in a variety of configurations. The components are able to connect, interact, or exchange resources (such as data) in some way, by adhering to a standardized interface. Unlike a tightly integrated system whereby each component is designed to work specifically (and often exclusively) with other particular components in a tightly coupled system, modular systems are composed of components that are "loosely coupled".

Modularity in SLDL is essential for structuring, reuse, hierarchical composition, maintainability, and applicability of different synthesis rules per structural entity (i.e. module). Modularity is also used for "separation of concepts" in SLDLs. The separation of computation and communication in modules/behaviors for computation and ports, interfaces and channels for communication [160] is one of the golden rules for system-level design. This separation of concepts enables independent design and refinement of computation and communication elements in the system. It also allows the integration of pre-designed blocks and the replacement of communication channels without affecting other modules/behaviors in the overall system.

Completeness refers to the ability to support all concepts found in embedded systems. For SLDSs this is the ability to describe different views (MoC, MoA, MoS, MoP) at different abstraction levels.

Orthogonality Orthogonality comes from the Greek orthos, meaning "straight", and gonia, meaning "angle". It has somewhat different meanings depending on the context, but most involve the idea of perpendicular, non-overlapping, varying independently, or uncorrelated. Orthogonality is a system design property which guarantees that modifying the technical effect produced by a component of a system neither creates nor propagates side effects to other components of the system. Typically achieved through separation of concerns and encapsulation, it is essential for feasible and compact designs of complex systems. The emergent behavior of a system consisting of components should be controlled strictly by formal definitions of its logic and not by side effects resulting from poor integration, i.e. non-orthogonal design of modules and interfaces. Orthogonality reduces testing and development time because it is easier to verify designs that neither cause side effects nor depend on them.

In SLDLs the separation of computation and communication enables orthogonality of these aspects. The replacement of a communication channel with another communication channel that implements the same interface in another way or with more implementation and timing details, does not influence the behaviors that use this channel. An example for orthogonality of data types is that all operators should work on all data types. After replacing a variable by a signal, all arithmetic operations that have been defined for the variable shall also be applicable to the signal.

Simplicity is the state or quality of being simple. It usually relates to the burden which a thing puts on someone trying to explain or understand it. Something which is easy to understand or explain is simple, in contrast to something complicated. The concept of simplicity has been related to truth in the field of epistemology. According to Occam's razor[4], all other things being equal, the simplest theory is the most likely to be true. For SLDs simplicity can be defined as

- easiness to explain and learn (i.e. based on a known programming paradigm),
- leaning on concepts known from programming languages,
- enabling reuse of behavioral code and algorithms (i.e. C code) and
- automatable refinement/model to model transformation in a single language (i.e. synthesis).

3.6.2 Program State Machines

Program State Machines (PSM) combine some of the essential formalisms used in the design of digital systems (cp. [160]). Figure 3.13 give a graphical overview of the different MoCs that are combined in a PSM. In the first part of this section a brief overview of the different MoCs from Figure 3.13, leading to the PSM definition, will be given. In the second part of this section a graphical notation for hierarchical PSM composition will be given. In the last part of this section the separation of behavior and communication using ports, interfaces and channels will be presented.

As programming language (see Section 3.6.3) in our PSM model a C++ subset will be defined in Section 5.3. The internal graph representation of PSM leaf behaviors will be presented in Section 3.6.3.

3.6.2.1 Program-States

A **Finite-State Machine** (FSM) is the most popular model for the description of control systems. An FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions.

Definition 3.6.2.1 (Finite-State Machine (FSM)):
A Finite-State Machine (FSM) $FSM = [S, I, O, F, H, s_0]$ *is*

$$S = \{s_0, s_1, \ldots, s_l\} \quad \textit{is a set of all states}$$

[4] Occam's razor (also written as Ockham's razor, Latin: lex parsimoniae) is the law of parsimony, economy or succinctness. It is a principle urging one to select among competing hypotheses that which makes the fewest assumptions and thereby offers the simplest explanation of the effect.

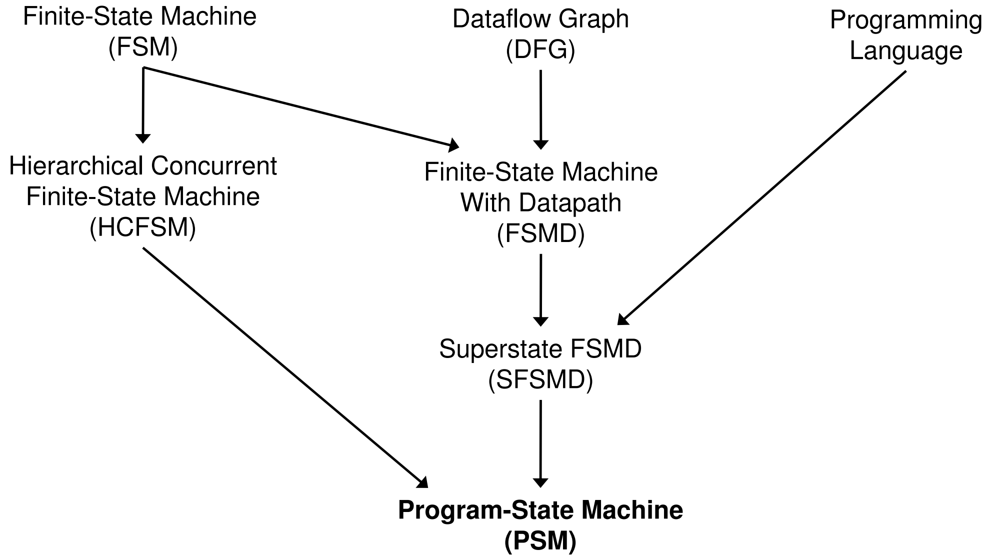


Figure 3.13: Overview of different traditional models unified in Program-State Machines

$$
\begin{aligned}
I &= \{i_0, i_1, \ldots, i_m\} && \textit{is a set of inputs}\\
O &= \{o_0, o_1, \ldots, o_n\} && \textit{is a set of outputs}\\
F &\colon S \times I \to S && \textit{is a next-state function}\\
H &\colon S \to O && \textit{is an output function}\\
s_0 &\in S && \textit{is an initial state}
\end{aligned}
$$

A moore-type FSM associates outputs with states $H\colon S \to O$*, as given above.*
A mealy-type FSM associates outputs with transitions $H\colon S \times I \to O$*.* □

An extension to eliminate the problem of the state and arc explosion is the introduction of concurrency and hierarchy. This model is called **Hierarchical Concurrent Finite-State Machine** (HCFSM) [169] and is implemented in the widely used StateCharts [194, 187]. States of a HCFSM can be decomposed into an FSM. Hierarchy, also known as OR-decomposition, can be decomposed into a flat FSM with the same number of states but more transitions. Concurrent states, also known as AND-composition, can be decomposed into a sequential FSM with more states and more transitions (cross product automaton construction).

A **Dataflow Graph** (DFG) is used for describing computationally intensive systems. Terms that are used to describe computations can be easily represented by a DFG. It consists of nodes that represent operations or functions. Directed arcs between nodes define the execution order. In the context of PSMs we consider dataflow graphs which are *homogeneous SDF graphs*.

Finite-State Machines with Datapath (FSMD) combine the features of the FSM and the DFG models since most real world systems consist of both control and computation. The FSMD is well suited for modeling hardware: Each state transition appears at a single clock cycle and the operations executed in each state can be interpreted as a set of register-transfer operations.

Definition 3.6.2.2 (Finite-State Machine with Datapath (FSMD)):
A Finite-State Machine with Datapath (FSMD) $FSMD = [S, I, O, V, F, H, s_0]$ *is*

$$
\begin{aligned}
S &= \{s_0, s_1, \ldots, s_l\} && \textit{is a set of states}\\
I &= \{i_0, i_1, \ldots, i_m\} && \textit{is a set of inputs}\\
O &= \{o_0, o_1, \ldots, o_n\} && \textit{is a set of outputs}\\
V &= \{v_0, v_1, \ldots, v_n\} && \textit{is a set of variables}\\
F &\colon S \times I \times V \to S && \textit{is a next-state function}\\
H &\colon S \to O \cup V && \textit{is an action function}
\end{aligned}
$$

$$s_0 \in S \quad \textit{is the initial state}$$

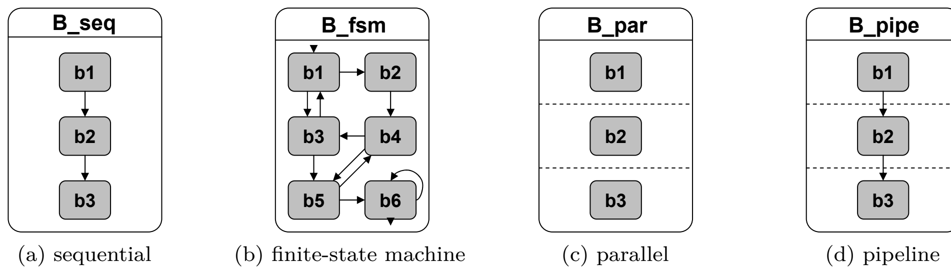
I, O, V may represent complex data types (i.e., integers, floating point, etc.). F, H may include arithmetic operations. H is an action function, not just an output function and describes variable updates as well as outputs. Complete system state now consists of current state, s_i, and values of all variables V. □

Merging the FSMD model with the concept of programming languages leads to the so-called **Superstate FSMD** (SFSMD). In this model a superstate does not represent exactly a single clock cycle as in the FSMD model, but any number of clock cycles which depend on the final implementation. Such a superstate can be specified by constructs of programming languages, as mentioned above.

Replacing the FSM model in HCFSMs by a SFSMD model leads to a HCSFSMD, or much shorter **Program-State Machine** (PSM) [185].

3.6.2.2 Hierarchical composition

A PSM consists of a hierarchy of program-states with each of them specifying a single mode of computation. At each point in time only a subset of program-states is active, and thus perform their computations. A composite program-state can be decomposed into either sequential or concurrent program-substates. The SpecC [160] language implements a PSM model of computation where program-states are called behaviors.



(a) sequential (b) finite-state machine (c) parallel (d) pipeline

Figure 3.14: PSM composite behaviors [150]

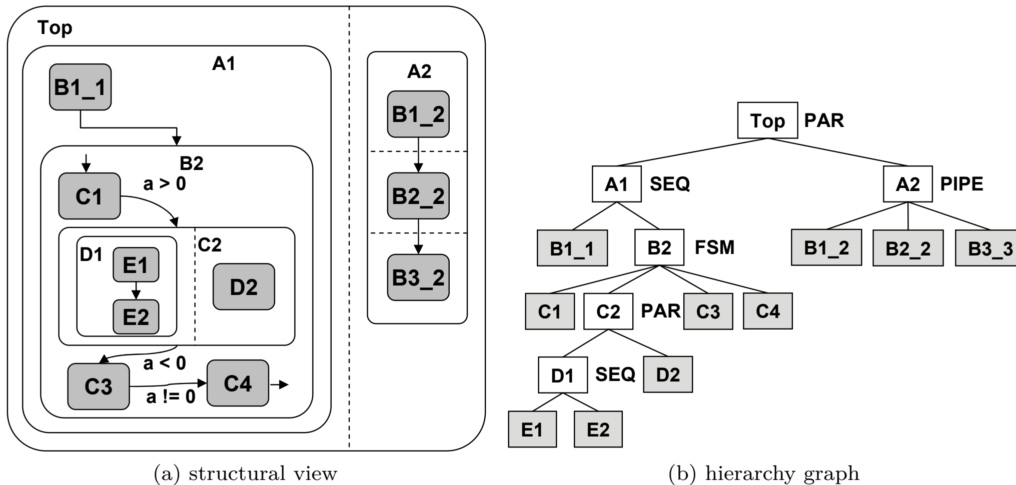
Figure 3.14 shows the SpecC composite behaviors. A sequential behavior describes a purely linear control flow between sub-behaviors. In Figure 3.14a leaf behavior `b1` is executed until it reaches its completion point, then leaf behavior `b2` is executed, and so on. `B_seq` is left when `b3` has finished its execution.

This kind of composition corresponds to the linear sequential execution as known from any imperative programming language. In the finite-state machine behavior in Figure 3.14b the execution order of the sub-behaviors depends on the evaluation of the transition arcs. After the initial state `b1` has been entered the successor state can be either `b2` or `b3`, depending on which transitions' guard expression (not shown in this figure) evaluates to true. `B_fsm` is left when `b6` has finished its execution and no transition can be taken.

In a concurrent behavior, all sub-behaviors become active whenever the parent behavior is entered. In Figure 3.14c the sub-behaviors `b1`, `b2` and `b3` are executed concurrently when `B_par` is entered. It is left when all concurrent sub-behaviors have finished their executions. This corresponds to the general FORK-JOIN pattern.

A combination of concurrent and sequential behaviors is the pipeline behavior. In general a pipeline describes some sort of stream processing in which the same sequence of operations is performed on a stream of data. When `B_pipe` in Figure 3.14d is entered `b1` starts its execution. When it is finished `b1` and `b2` execute in parallel until both of them are finished. It the next step `b1`, `b2` and `b3` execute in parallel until a certain stop condition evaluates to true. Otherwise a pipeline behavior runs forever.

Figure 3.15 shows how different composite behaviors can be used to assemble a complex hierarchical PSM.



(a) structural view (b) hierarchy graph

Figure 3.15: Hierarchical PSM

3.6.2.3 Communication

Considering communication in the design of embedded systems along with a strict refinement process towards a physical implementation model the following basic principles can be postulated:

(a) Separation of communication and computation.

(b) Declaration of abstract communication primitives.

(c) Enable custom communication implementation on different levels of abstraction.

SpecC provides channels for the explicit description of communication and thus enables the separation of communication and computation. A channel implements a certain interface that defines which communication primitives are provided. These abstract communication primitives can be used by behaviors that represent the computational parts of the design. Communication is initiated on ports which can be part of behavior. Only ports whose interface type matches with the interface implemented by a channel can be bound together and therefore establish a communication link. SystemC has adopted this design pattern directly from SpecC.

Figure 3.16 gives a motivation for the separation of computation and communication.

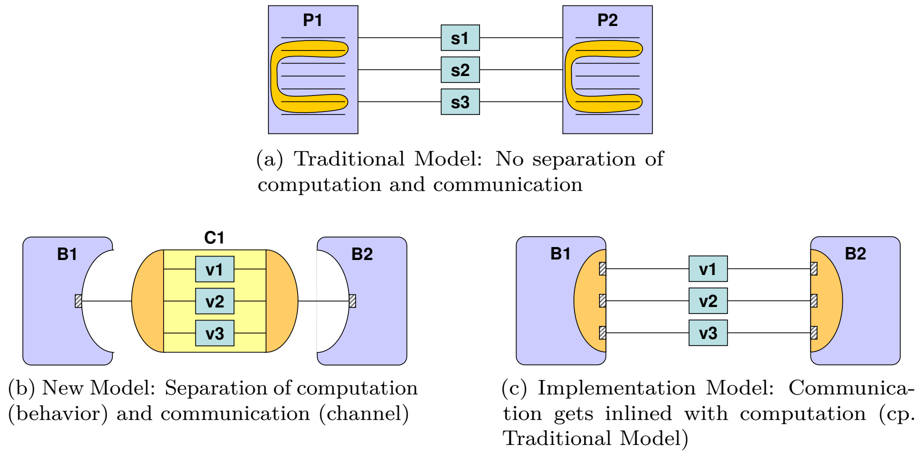
The traditional model of a block diagram, such as in VHDL or Verilog, is shown in Figure 3.16a. Two processes, $P1$ and $P2$, are communicating via signals $s1$, $s2$ and $s3$. By assigning values to the signals, according to some predefined protocol, the processes can communicate and exchange data. As a consequence $P1$ and $P2$ contain code for both communication and computation. The communication part in Figure 3.16a is highlighted, because communication and computation are mixed in the code and it is not possible to automatically change the communication protocol when design constraints change. At the same time it is also impossible to automatically switch to a new algorithm to perform the computation in a different way (i.e. replacing a software with a hardware algorithm).

In the model in Figure 3.16b communication and computation are separated. Computation is encapsulated in behaviors, and communication is encapsulated in channels. The computation is encapsulated in the behaviors $B1$ and $B2$, and the communication is contained in the channel $C1$. The channel $C1$ encapsulates the communication as functions, like *send* and *receive*. These functions define the interfaces of the channel. The channel also encapsulates the communication media, i.e. the variables $v1$, $v2$ and $v3$. On the other hand, the behaviors only contain computation. In order to communicate, the behaviors call the functions of the channels on ports connected to the communication service providing channel.

As a result of the separation of communication and computation, the model in Figure 3.16b supports "plug-and-play". The communication protocol can be exchanged by use of another channel with compatible interfaces, whenever this is desirable in the design process. In the same

manner, the behaviors can also be exchanged with others, without affecting the communication protocol.

In the final implementation model, channels are reduced to variables or signals (representing wires), like in the traditional model (see Figure 3.16c). For the implementation of a channel, its functions are inlined into the connected behaviors and the encapsulated communication media are exposed. After inlining, the channel $C1$ has disappeared. The encapsulated variables $v1$, $v2$ and $v3$ are exposed and the communication protocol is integrated into the behaviors $B1$ and $B2$.



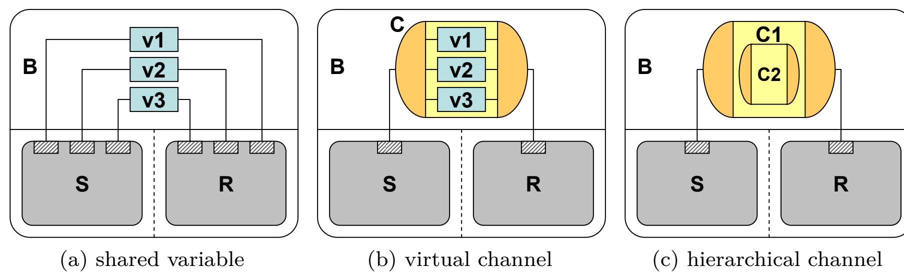
(a) Traditional Model: No separation of computation and communication

(b) New Model: Separation of computation (behavior) and communication (channel)

(c) Implementation Model: Communication gets inlined with computation (cp. Traditional Model)

Figure 3.16: Separation of communication and computation using channels [160]

Figure 3.17 gives an overview of different communication methods. In Figure 3.17a shared variables `v1`, `v1` and `v3` are bound to ports of behavior `S` and `R`. Ports have an explicit direction *in*, *out*, and *inout*, that are used to control the direction of data-flow. This model of communication corresponds to shared memory communication style. In this kind of communication race-conditions can appear and need to be avoided through careful design (e.g. all ports of `S` are output ports and all ports of `R` are input ports), or explicit synchronization between behaviors `S` and `R` before accessing shared variables. Whenever possible shared variables between parallel behaviors should be replaced by an explicit channel with access control and synchronization capabilities.



(a) shared variable (b) virtual channel (c) hierarchical channel

Figure 3.17: Overview of communication methods [160]

In Figure 3.17b a virtual channel that encapsulates the communication using shared variables (and events) through a user-defined interface is shown. An example for such a virtual channel is a CSP-like "double handshake channel". This kind of communication corresponds to message passing communication where both, the sender and the receiver, need to synchronize before data is copied from the sender to the receiver. An extension of a virtual channel is hierarchical channel as shown in Figure 3.17c. This type of nested channel to contains virtual sub-channels and is commonly used for the description of layered protocol stacks as used in modern shared bus or network on chip communication infrastructure.

3.6.3 Sequential program representation

As shown in Figure 3.13, programming languages are an important ingredient of Program State Machines for the description of functionality in leaf behaviors.

Programming languages in general provide a heterogeneous model that supports modeling of data, activity, and control. Programming languages can be classified into two basic paradigms: imperative and declarative. Over the last decade imperative programming languages have become far more accepted than declarative ones. Today, most imperative programming languages support object-oriented features like encapsulation of data and operations, inheritance, and polymorphism. Thus, data is modeled by basic types (e.g. integers and reals), composite types (arrays or structs of basic types) or objects. Activities are modeled by statements, while larger activities can be structured by functions and procedures. Activities on objects are initiated through method calls. Control flow is described by control constructs that specify the order in which activities are to be performed. In imperative languages these are sequential composition (often denoted by a semicolon), branching (`if` and `switch` statements), looping (`while`, `do-while` and `for` statements), and routine or method calls.

This section introduces basic notations that are used to represent programs in this work. Moreover, the structure of considered programs will be defined. All definitions used in this section are reproduced from [139]. A program P consists of variables, expressions (computing a value from the operands), comparisons, assignments and control structures (loops and branches). Depending on the control flow, a given sequence of expressions is split into basic blocks.

Definition 3.6.3.1 (Basic Block):
The control flow of a program is defined by jumps which are intra-procedural branches, and calls, which are inter-procedural branches. These branches divide the program into basic blocks. Control flow enters basic blocks at the beginning and leaves them at the end, without the possibility of branching except for the end of the basic block.

Let V be the set of basic blocks. For a given program P this set must be finite: $|V| < \infty$. □

Definition 3.6.3.2 (Routines):
Structuring a program into re-usable (usually parametrized) smaller pieces is done by routines. In literature the terms "function" and "procedure" also occur, but we define "routine" as a generalized program substructure also to avoid confusion with mathematical functions.

Let R be the set of routines of P and let r_0 be the routine to be invoked upon start of P. In other words, r_0 is the main routine of P.

Every basic block belongs to exactly one routine. The function $routine\colon V \to R$ associates each basic block with its routine. V_f describes he set of basic blocks of each routine: $V_f = \{v \in V | routine(v) = f\}$.

Each routine has exactly one basic block that is the first to be executed upon invocation. This basic block is called start node. The set $Starts \subseteq V$ contains all start nodes of P, one for each routine. The function $start\colon R \to Starts$ associates the start node with its routine.

Another important set of basic blocks is constituted by those that contain routine invocations. These basic blocks are called call nodes. The set $Calls \subseteq V$ contains all call nodes of P. Call nodes may be associated with more than one start node, if there is more than on possible routine to be invoked. This happens for computed calls. The following function associates call nodes with their invoked start node, is defined for all nodes, and returned $\{\}$ for non-call nodes:

$$\begin{aligned} target \quad : \quad & V \to \wp(Starts) \\ & v \mapsto \{v' \in V | v \text{ invokes } v'\} \end{aligned}$$

□

Definition 3.6.3.3 (Control Flow Graph):
Each routine has it own control flow graph, consisting of nodes that are basic blocks, and edges representing the control flow between the blocks. Let $CFG_f = (V_f, E_f)$, $f \in R$ be the control flow graph of routine f.

A control flow graph has exactly on start node $start(f)$ via which all control flow enters routine f. For a path in a graph G, e.g. CFG_f, from node v_1 to v_2, we will write $v_1 \to_G^* v_2$.

If control flow has several alternative possibilities to continue at run-time after a given basic block, i.e., if a node in a graph has several outgoing edges, this situation will be called a branch. Branches in control flow graphs will be called jumps. □

Definition 3.6.3.4 (Call Graph):
A call graph connects call nodes and start nodes. $Calls$ and $Satrts$ constitutes the nodes and $target$ restricted to $Calls$ defines the intra-procedural edges. The linkage between start and call nodes is established by adding edges from start to call nodes for each routine.

Formally, a call graph is defined as: $CG = (\hat{V}, \hat{E})$, *with*

1. *the nodes* $\hat{V} = Calls \cup Start$,
2. *and the edges* $\hat{E} \subseteq \hat{V} \times \hat{V}$, *where* $\hat{E}$ *is defined as:*

$$\begin{aligned} \hat{E} \quad &:= \quad \{(c, s) \colon c \in Calls, s \in target(c)\} \cup \\ \textstyle\bigcup_{f \in F} \quad & \{(s, c) \colon s \in Starts, c \in Calls \colon \exists s \rightarrow^{*}_{CFG_f} c\} \end{aligned}$$

□

It is required that call nodes have exactly one incoming edge in the CFG. This can be ensured by inserting additional empty nodes for the call nodes that contradict this requirement. Together with the definitions above, each call node c also has exactly one outgoing edge in the CFG. In the CG, c also has exactly one incoming edge (from the start node) and possibly several outgoing edges (defined by $target(c)$).

```
1  void a() {
2     ...   // basic block b1
3  }
4  void b() {
5     a(); // invocation c3
6  }
7  int main() {
8     a(); // invocation c1
9     b(); // invocation c2
10 }
```

Listing 3.2: Example C code for call graph in Figure 3.18

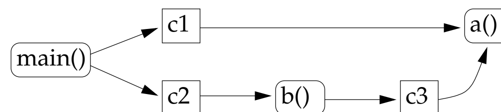


Figure 3.18: Call Graph (CG) of Example Listing 3.2. Start nodes are labeled with the name of the routine and a pair pf parentheses, call nodes are labeled as shown in the comments in the C source code. The defined CG does not contain return edges, only call edges [139].

Definition 3.6.3.5 (Loops):
The term "loop" will be used for a natural loop as defined in [75]. A natural loop has the following basic properties:

1. *A natural loop has exactly one start node which is executed every time the loop iterates. This node is called header.*
2. *A natural loop is repeatable, which means that there is a path back to the header.*

□

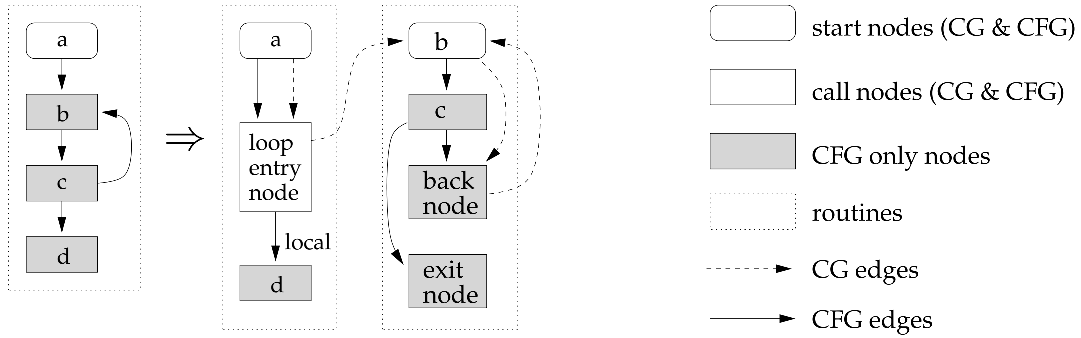


Figure 3.19: Control Flow Graph (CFG) modifications by loop transformation. Loop transformation introduces a ew routine and new call nodes for each loop and transforms the loop into a recursive routine. Dashed lines represent edges in the call graph, which are introduced by this transformation [139].

Loops and recursion are handled uniformly by transforming all loops into recursive routines by making the loop body a routine on its own and inserting inter-procedural edges accordingly.

Another imposed restriction for control flow graphs is, that apart from loops there must be no other cycles. This restriction is not too strict, since well-done hand written sequential code, without explicit jumps using `goto` statements, fulfills this requirement. The use of this restriction in motivated in the unbounded run-time of control flow cycles. When a path analysis searches the maximal run-time, loops must be bounded with maximal iteration counts in order to make the problem solvable. For recursive function calls in this context only tail recursion is allowed, since tail recursion elimination allows to replace recursive function calls through loops. But anyhow, to enable analysis the recursion depth will limit the number of loop cycles.

So we assume that aft loop transformation, there must be no cycles in the control flow graphs at all. All cycles must have been moved to the call graph and marked as loops. Let L be the set of loops of a program P. Since loops are converted into recursive routines, it holds that $L \subseteq R$. The header of a loop is the start node of the loop.

Definition 3.6.3.6 (Loop Header):
The header fo a loop $l \in L$ is defined as:

$$\begin{aligned} header \quad : \quad & L \to Start \\ & l \mapsto start(l) \end{aligned}$$

□

Definition 3.6.3.7 (Entry and Back Edges):
The functions

$$\begin{aligned} entries \quad : \quad & L \to \wp(\hat{E}) \\ back \quad : \quad & L \to \wp(\hat{E}) \end{aligned}$$

assigne to a loops its entry and back edges. □

Definition 3.6.3.8 (Minimum and Maximum Loop Iteration Count):
The number of iterations of loops is specified using two functions. One for the minimum iteration count and one for the maximum. Iteration bounds are defined for each entry of the loop and, therefore the functions take the loop entry node as input argument.

Let l be a loop and $e \in entries(l)$ one of its entry edges:

- $n_{min}(e)$ *defines the minimum loop execution count per entrance of l via e*
- $n_{max}(e)$ *defines the maximum loop execution count per entrance of l via e*

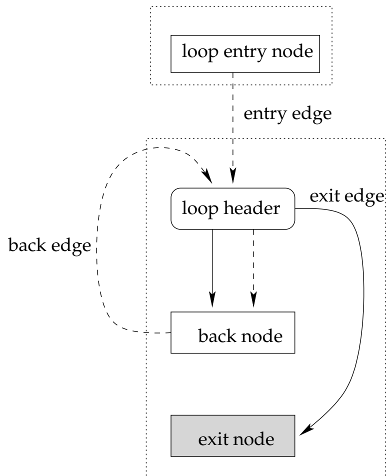


Figure 3.20: A simple loop with all the important edges. Dotted lines and white nodes are in the call graph, the other items are part of the control flow graph [139].

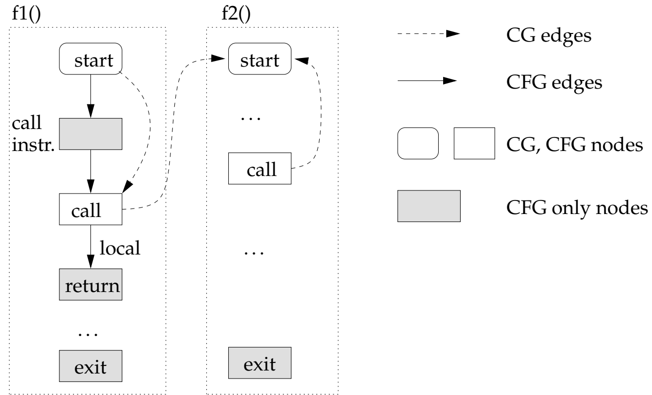


Figure 3.21: *CFG* and *CG* of a call of a recursive routine. This figure clarifies the use of local edges ans shows that there are no return edges (e.g. from a note in routine `f2()` back to the call node in `f1()`) [139].

□

Definition 3.6.3.9 (Artificial Empty Nodes):
For providing special hooks for analysis and timing annotation we introduce four additional artificial empty nodes as specific locations in the CFG and CG. At each routine invocation, two additional nodes are inserted:

- *call node: liked with the start node in the CG. The actual call instructions are located in the block before the call node.*
- *return node: that indicate the return of a called routine.*

Routines are augmented with the following empty nodes:

- *start node: routines begin with a start node*
- *exit node: returning control flow to the caller is gathered in a unique exit node.*

□

Definition 3.6.3.10 (Alternative Control Flow and Edge Types):
Alternative control flow occurs at two levels; in the control flow graph, where if-then-else statements are the most common example, followed by switch statements, and in the call graph, where function pointers are the most common example. Virtual function calls are a special case of function pointers.

To handle alternative control flow in CFG, there are different types of edges. These different types of edges can be used to express different execution times, since jumps for instance have different execution times for different types of edges.

The edge type is defined as a function that assigns a type to each edge:

$$type \quad : \quad E \rightarrow \{normal, false, true, local\}$$

normal edge: *outgoing edge of basic blocks whose control flow exits the block without alternatives (i.e. without a branch).*

false edge: *for a conditional jump, this marks the edge that is taken if the branch is not taken (also known as fall-through edge). At each block there is maximally one of these edges.*

true edge: *for a jump, this marks possible branch targets of the jump.*

local edge: *represents the control flow after a call. This edge is inserted after each call node, because the graph will not contain flow information about routine returns.*

For switch tables, there may be a number of true edges, one for each possible branch target. Alternative control flow in the call graph is marked in the same way, by using multiple outgoing edges from a call node to several start nodes. □

CHAPTER 4

Related Work

4.1 Introduction

The OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) System Level Design Language (SLDL) and methodology has been developed in three successive ICT research projects (see also Figure 1.1):

ODETTE: **O**bject-oriented co-**DE**sign and functional **T**est **TE**chniques [224]

ICODES: **I**nterface- and **CO**mmunication-based **D**esign of **E**mbedded **S**ystems [223]

ANDRES: **AN**alysis and **D**esign of run-time **RE**configurable, heterogeneous **S**ystems) [220]

It provides a design methodology, modeling & simulation library and synthesis tool for communication dominated hardware/software systems. It allows the designer to start with an abstract functional executable specification using communication objects between parallel processes. Designs written in OSSS can be systematically refined and synthesized to various FPGA-based target architectures and technologies.

The OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) modeling & simulation library is based on SystemC™ [13]. This library enables the combination of object-oriented concepts known from parallel object-oriented software design with digital hardware/software design. While raising the level of abstraction by introducing object-orientation and transaction level modeling (TLM) techniques, OSSS aims to obtain well-defined synthesis semantics for all available modeling elements allowing automatic and system synthesis of OSSS models into

- an overall hardware platform description,
- custom software and communication drivers in C++ and
- custom hardware modules in VHDL or SystemC models at Register Transfer Level (RTL).

The resulting hardware platform, software and hardware module descriptions are further processed using well-established software cross-compilers and RTL-to-gate-level synthesis tools.

Historically, OSSS has been mainly motivated by the use of the object-oriented features of C++ in synthesizable SystemC models. As the term "class library" suggests, SystemC internally makes use of the object-oriented features of C++, e.g. classes, inheritance, polymorphism, communication by method calls, etc. However, these features are not available to the designer at the same extent when writing synthesizable models. Before synthesis these usually need to be manually refined to either behavioral or register-transfer level descriptions, constructed from modules with ports, connected via signals with behavior described as explicit or implicit state-machines using processes.

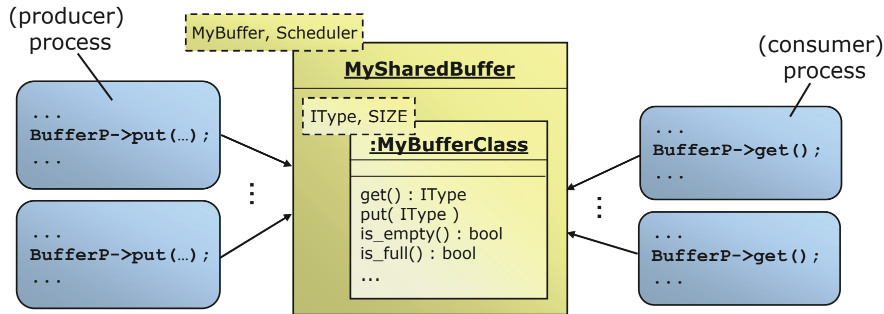


Figure 4.1: Accessing a Shared Object from multiple producer and consumer processes.

Even when neglecting synthesis, the use of bare variables and method calls between parallel processes in pure simulation models can be problematic[1].

This is where OSSS applies. By allowing object-oriented features, like classes and objects, inheritance, method calls and operator overloading with a well-defined synthesis semantics, the level of abstraction and the expressiveness are increased.

Another challenge that arises during the integration of hardware and software is the hardware/software communication. When using SystemC to develop an executable specification of a hardware/software design on a functional level, it is impossible to synthesize it for a certain hardware platform without extensive manual refinement. Especially the implementation of the hardware/software communication interface is a very error prone and time-consuming task. In order to avoid this manual refinement effort, or to keep it as low as possible the OSSS methodology defines three modeling layers (*Behavioral Layer*, *Application Layer* and *Virtual Target Architecture Layer*) with dedicated modeling elements and well-defined mapping and refinement rules between elements in these layers. Finally, the *Virtual Target Architecture Layer* description can be synthesized for a specific target platform. Up to now, only a synthesis back-end for Xilinx FPGAs exists.

Communication modeling is one of the key aspects in the design of complex custom embedded hardware/software systems. Methodologies and techniques that help to improve the modeling and synthesis of communication, especially across the hardware/software boundary can also significantly improve the whole design process.

System-Level Design Languages, like SpecC and SystemC enable the separation of computation and communication by providing the concepts of ports, interfaces and channels. A channel can be used to encapsulate and abstract from the details of a certain communication - for instance, a certain protocol -, and allows processes to communicate and exchange data with each other, based on interface method calls. This concept already has led to a new kind of design methodology, called transaction level - or transaction based - modeling [131].

The channel concept provides for modeling communication at a higher level, and clearly separates communication and computation. This makes channels very useful for creating and refining models for simulation. Since channels do not possess a synthesis semantics in general, manual channel refinement to a signal level representation or by replacing channels with pre-designed synthesizable IP components is currently applied.

In this work we use an alternative concept for SystemC channels, based on so-called Shared Objects [135, 97]. Like any other object, it may specify a set of public and private or protected methods, while public methods define its interface. This enables processes to call implemented services on the Shared Object. These user-defined services enable the implementation of inter-process communication, event notification and data exchange. Figure 4.1 illustrates the implementation of a shared FIFO buffer. Like channels in SystemC, Shared Objects are using the concept of port interface binding and Port-Interface-Calls to enable structural decomposition (e.g. connecting processes and Shared Objects through a hierarchy of modules).

A Shared Object possesses built-in and customizable mechanisms for handling concurrent accesses, a clear communication refinement and synthesis semantics, and exhibits a timed

[1]Consider, for example the case when two different processes simultaneously call methods of the same object, which may lead to an undefined state - similar to writing shared variables from different processes.

behavior during simulation. Handling concurrent accesses is realized by means of a scheduler that can be specified by the user for each Shared Object. The scheduler determines which client process is granted access to the Shared Object in the case of concurrent requests. All other requesting clients are blocked meanwhile. Consequently, accesses are mutual exclusive. In addition, this is supported by a guard mechanism which allows to associate a side-effect free Boolean expression[2] - the so called guard - with a member function[3]. Client processes calling a member function (i.e. a service provided by the Shared Object) whose guard evaluates to false are ignored for scheduling and the call is effectively blocked until the guard evaluates to true due to Shared Object state change, effectively caused by another service call. The timed behavior of Shared Objects gives the designer an early and realistic impression on the temporal behavior of the modeled system during simulation even before performing synthesis to a signal level representation.

In Section 4.2 a comparison with previous work towards object-oriented communication synthesis this work builds on is given. In the following sections a description of the state-of-the-art and classification of this work with respect to the following topics is given:

- Section 4.3 provides an overview of relevant object-oriented communication concepts known from parallel distributed system design, applied to embedded system design and used in Electronic System-Level Design (ESL). Covered work is spanning from a common hardware/software object model, actor- and active object-based design, Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI) techniques. This section targets a classification and comparison of the object-oriented communication techniques provided by the Shared Objects with other work.

- Section 4.4 provides an overview of past and recent work on SoC platform communication modeling. It covers the entire spectrum form Transaction Level Modeling (TLM), Communication Architecture Functional Modeling, to Bus Accurate and Cycle-Accurate Modeling. This section targets a classification and comparison of the proposed RMI and OSSS Channel approach with existing work.

- Section 4.5 provides an overview of relevant SoC communication synthesis techniques. Concerning communication synthesis from abstract channel and protocol specifications to RTL, several works exist in literature. This overview focuses on C and C++ based approaches only. This section targets a comparison of the Shared Object, RMI and OSSS Channel synthesis approach with existing work.

- Section 4.6 gives an overview of entire ESL synthesis methodologies and existing frameworks available in the research landscape. This section targets a classification and comparison of the overall OSSS methodology with other approaches.

Finally, in Section 4.7 the contribution of this work, with respect to the analyzed related work is given and discussed.

4.2 Previous Work

4.2.1 Objective VHDL

In [162] the generation of static digital circuit structures for application specific integrated circuits from object-oriented models is described. It contains synthesis concepts for classes with attributes and methods, inheritance of classes, message exchange between objects, and polymorphism. This work describes synthesis concepts and optimization techniques based on a meta model of object-oriented system descriptions. This also contains a synthesis rule for VHDL shared variables, which are not included in the VHDL synthesis subset. Figure 4.2 depicts the proposed hardware implementation of a shared variable, a predecessor of the Shared Object. As a result, Objective VHDL, an object-oriented variant of the hardware description language VHDL, has been implemented. In addition to the definition of the Objective VHDL language, a

[2] implemented as Boolean expression on the state variables of the Shared Object
[3] the term "member function" is used as synonym for a method or service, implemented in a class

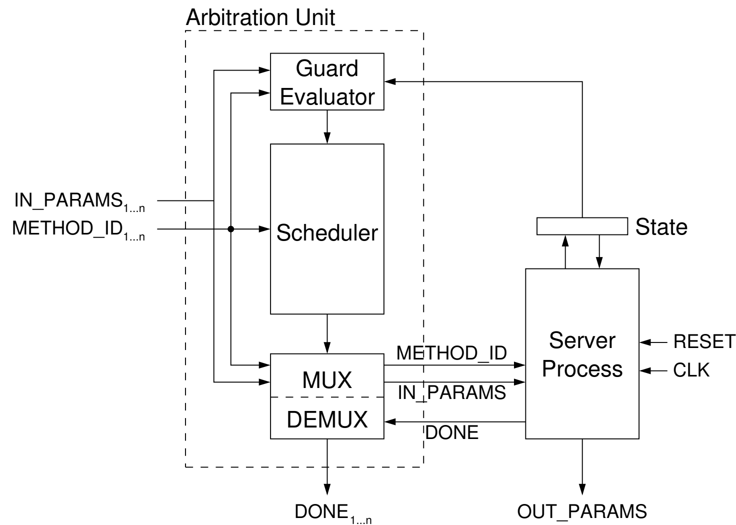


Figure 4.2: Shared variable hardware implementation as proposed in [162] (Source: [97])

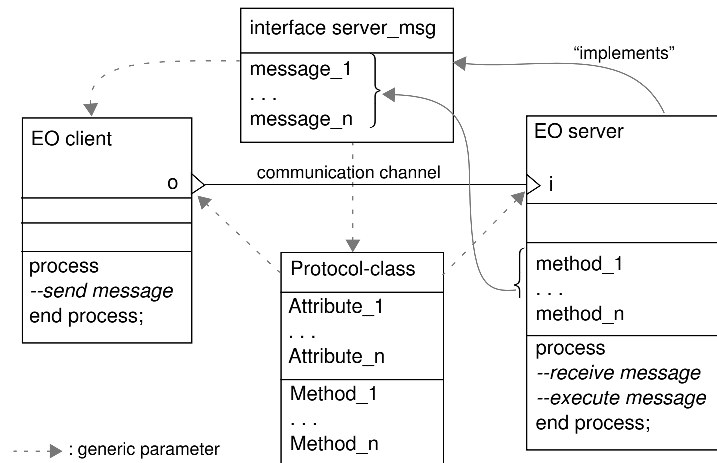


Figure 4.3: Objective VHDL+ generic communication model [153] (Source: [152])

translation tool for generation of industry standard VHDL source code from a synthesizable Objective VHDL model has been proposed.

4.2.2 Objective VHDL+

In [153] an extension for Objective VHDL to enable a consistent communication design for the structural object-oriented hardware design. As a result generic method call concept based on a message passing protocol has been proposed. For the separation of computation and communication, dedicated Channel Objects using interface classes and Channel Ports have been introduced to Objective VHDL, resulting in the Objective VHDL+ extension. Figure 4.3 depicts the proposed generic communication model where the server implements a method interface, which can be directly used by the client. The protocol class implements the communication channel, connecting client and server. It maps a client's method call to the physical communication channel's signal interface. Finally, a synthesis of the newly integrated object-oriented modeling elements to standard VHDL has been proposed. Figure 4.4 shows the signal level implementation of the client to server communication after synthesis.

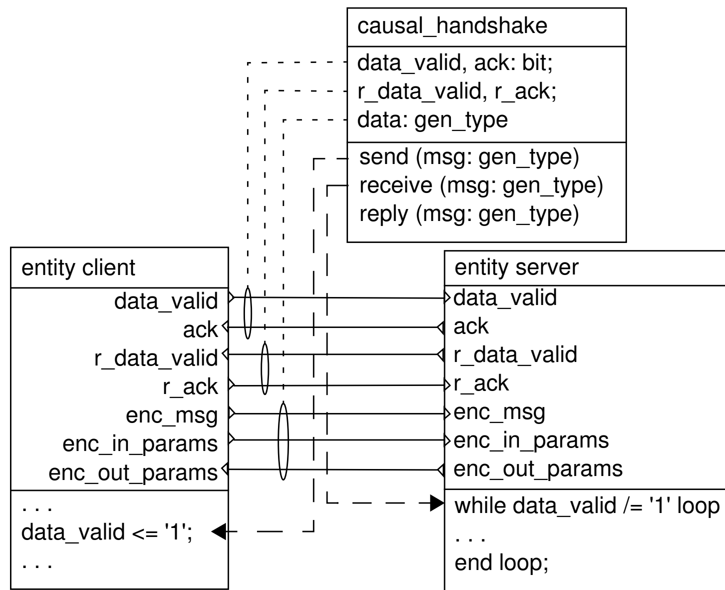


Figure 4.4: Objective VHDL+ generic communication model synthesis result[153] (Source: [152])

4.2.3 SystemC-Plus

In [135] present an approach based on the ideas of [162] for object oriented hardware design and synthesis based on SystemC. This work essentially distinguishes between two basic principle approaches to object-oriented hardware design: *Structural* vs. *Data Type* based object-orientation: *Structural* approaches model whole hardware components/entities as concurrent objects, while *Data Type* based approaches make use of object-orientation as a way to create new user-defined data type. To avoid the problem of inheritance anomaly[4] [135] focuses on data type based object-oriented modeling only.

In particular, [135] proposed an extension of the SystemC synthesizable subset [33] to support:

- declaration of classes and creation of new classes by means of inheritance from already existing classes,
- deriving classes from multiple and virtual parent classes,
- declaration of data members of scalar and complex type including class types and array types,
- declaration of member functions and operators,
- redefining member functions and data members in derived classes,
- declaring constructor methods and
- declaring class templates with scalar and type parameters.

[4]The fundamental object-oriented concept of inheritance can be hardly combined with the concept of concurrent objects.

"The Inheritance Anomaly is a failure of inheritance to be a useful mechanism for code-reuse that is caused by the addition of synchronization constructs (method guards, locks, etc.) to object-oriented languages. When deriving a subclass through inheritance, the presence of synchronization code often forces method overriding on a scale much larger than when synchronization constructs are absent, to the point where there is no practical benefit to using inheritance at all." [92]

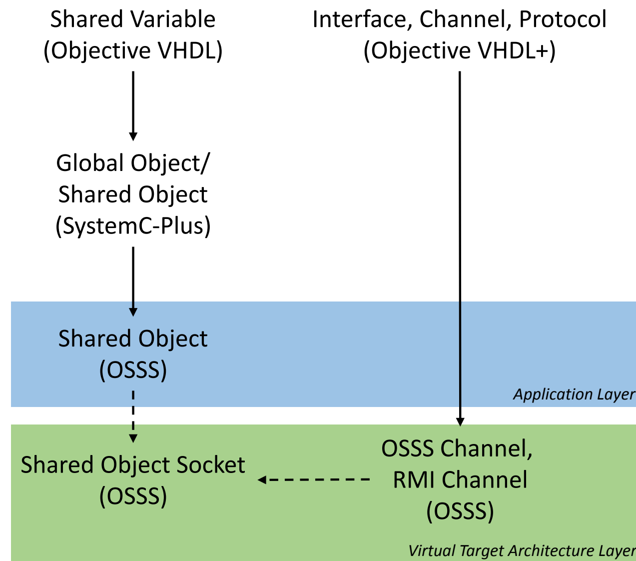


Figure 4.5: Evolution and relation to previous work

Furthermore, some limited support of polymorphism, without the use of pointers, has been introduced, called *tagged object*. It is a self-contained object, with an own state space, which means it is not just a reference to an object, like a pointer. Thus making reference resolution for a polymorphic object at run-time obsolete.

Finally, the concept of a Global Object has been introduced. A global object is an object, which is declared as member of a SystemC module. Like any other object, a global object may specify a set of methods, which forms its interface. This allows processes to communicate and exchange data via global objects based on method calls. Global Objects are a predecessor of Shared Objects as extended in this work.

In [97] targets performance optimizing hardware synthesis of Shared Objects in hardware specifications (based on Global Objects from [135]). It is based upon an earlier proposal for the hardware implementation of shared variables [162] (see Figure 4.2). The main problem addressed is the research and development of alternative implementations for Shared Objects, which are more efficient. I.e. provide a better performance/area ratio, and/or provide a higher absolute performance compared to [162].

The proposed optimizations target the delay, which is caused when accessing a Shared Object due to the necessary synchronization between the calling process and the object. This has been realized by optimizing the implementation of a Shared Object itself, such that the speed at which accesses are handled is increased, and by optimizing the communication between Shared Objects and accessing processes.

4.2.4 Discussion

This work builds on top of the foundation of [162, 153, 97] and combines Shared Objects from [97] with the generic object-oriented communication model of [153]. Figure 4.5 visualizes the evolution of the discussed concepts and integration in the OSSS Virtual Target Architecture Layer.

In previous work, the concepts of Shared Variables/Objects and custom communication channels have been strictly separated. While previous work only focused on the object-orient description of custom hardware, integration of Shared Objects with its client processes has been implemented through fixed point-to-point connections. These point-to-point connections have been implemented using a simple protocol working on a parallel set of wires that match the bit width of the largest method/service. While [97] considered optimizations of the Shared Object access protocol it did not allow for customizing or constraining the communication protocol between client processes and the Shared Object.

The communication channel concept, as presented in [153] offers the possibility to define user-defined point-to-point channels and protocol implementations on a physical channel (i.e. bunch of wires/signals). The development of such an customizable point-to-point channel has been driven by the requirement to enable a stepwise mapping to a constrained set of communication resources (e.g. total number of allowed wires) and a custom protocol definition (e.g. handshake channel, FIFO).

Since this work addresses the challenge of object-oriented hardware/software communication, the idea was to combine both, the high-level concept of Share Objects as Application Level communication objects and the refinement of client to Shared Object connections. This becomes even more necessary, when targeting SoC platforms with pre-existing communication IP, such as buses or crossbar channels, which need to be used to physically bridge hardware/software boundary.

As a result, this work has extended the Shared Objects of [97] to Application Layer message passing channels, that fully integrate into the SystemC and SpecC communication channel methodology (e.g. using explicit port to interface bindings). Furthermore, Application Layer Shared Objects are explicitly mapped to Shared Object Sockets on the Virtual Target Architecture. These sockets enable the explicit connection with SoC communication resources, such as buses or user-defined point-to-point connections.

For this purpose the generic communication channel concept of [153] has been extended be OSSS Channels to cover topologies beyond point-to-point connections (e.g. shared buses and crossbar switches). The concept of Remote Method Invocation enables a transparent mapping of the Application message passing communication to the physical channel implementation inside OSSS Channels.

Dedicated simulation models for OSSS and RMI channels enable to analyze and timing behavior during simulation. In previous work [97] the timing behavior of a Shared Object has only been accessible after synthesis to VHDL. This extended simulation model enables faster design space exploration and a true single language approach.

4.3 Object-Oriented Communication Concepts in ESL Design

Beyond previous work [162, 153, 135, 97] on object-oriented design and synthesis of custom hardware, the principle of object-oriented design has been applied to ESL design in general. In this section, related work in the area of object-oriented communication concepts applied to ESL design is presented and discussed.

4.3.1 OOCL

In [179] Vahid et. al. propose the first idea of an object-oriented communication library for hardware/software co-design. This library aims to encapsulate communication protocol specific routines and functionality into channel classes. These channel classes provide generic send and receive methods to be used by the user program. In this work, the implementations of a serial point-to-point protocol and a shared bus protocol are presented. Together with the classes, the library also contains VHDL code for the hardware communication interface, called port. Multiple channels (i.e. instances of channel classes) can use the same hardware port one after another. The send and receive methods a non-interruptible.

While providing a separation of user program computation and communication through encapsulation of the communication protocol implementation in so-called channel classes. Beyond this encapsulation, the approach does not take further advantage of object-oriented features, like serialization and de-serialization of user-defined classes.

From the methodological point of view, no explicit separation of port, interface and channel is provided, thus making use in a system-level model with gradual communication channel refinement impossible.

4.3.2 CHSOM

In [158] an extension to the traditional component-based development model to include both hardware and software has been proposed. In the proposed Common Hardware and Software Object Model (CHSOM) the (Distributed) Common Object Model (COM/DCOM) concept is generalized towards hardware, software, and hardware/software objects. The Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft. It is used to enable inter-process communication and dynamic object creation in a large range of programming languages.

In CHSOM, each object is described by a set of interfaces (at least one), a contract (representing whether the component is for hardware or software, information about specification and implementation, and information for software and hardware simulation), and a standard messaging/invocation/linking mechanism. All of these features are implementation, and caller independent: Each component and its interfaces are specified using an interface description language (IDL) that is implementation language neutral. The binary communication interface is well defined locally and across a network.

Although this is an interesting approach, there is no evidence that the proposed approach has been successfully implemented and evaluated.

4.3.3 Actor-oriented

The actor model is a generic model of computation where actors are the universal primitives of concurrency [201]. In actor-oriented design, everything is an actor, similar to object-oriented design where everything is an object. The main difference between object- and actor-oriented designs is how concurrency is treated and expressed. In object-oriented design, parallelism needs to be expressed explicitly, while actors are implicitly parallel.

In general, an actor is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors,
- create a finite number of new actors,
- designate the behavior to be used for the next message it receives.

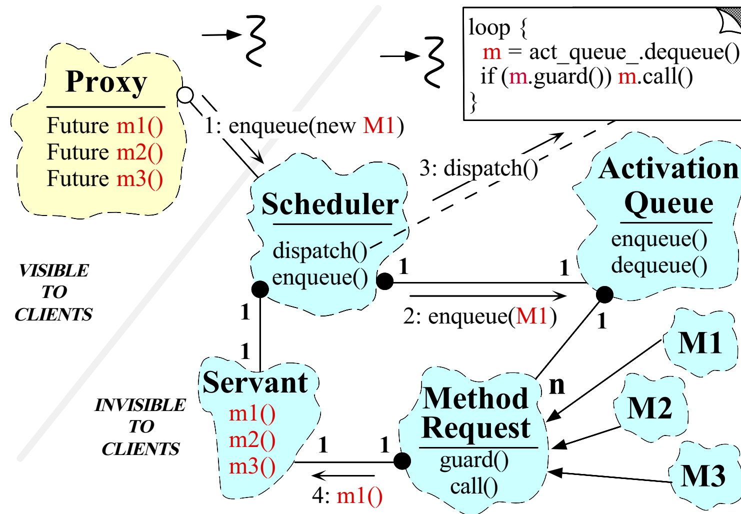
Actor-oriented design has been successfully applied to the design of distributed systems [196] and to the design of embedded hardware and software systems [138, 31]. Actor-oriented platforms, such as Simulink [228] or Ptolemy II [134], abstract aspects of program-level platforms, such as Java, C++, and VHDL.

From a methodical standpoint, actor-oriented design enables the separation of the actor definition language and the actor composition language, enabling highly polymorphic actor definitions and design using multiple models of computation as demonstrated in [134].

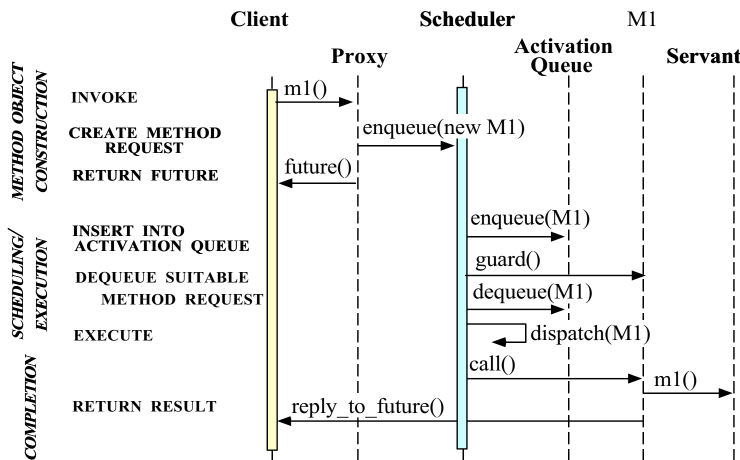
For equipping object-oriented languages with the concept of concurrency, object-oriented and actor-oriented techniques have been combined. In [161] a survey of concurrent object-oriented Languages is provided. It furthermore describes the challenges and difficulties when combining both techniques (inheritance anomaly is one of these). For this reason, most practically applied concurrent object-oriented languages put strict constraints on the combination of actor- and object-oriented features.

The Application and Virtual Target Architecture Layer models, proposed in this work, make use of a combination of actors and objects. After mapping Shared Objects on the Virtual Target Architecture Layer, client and Shared Object are running in two independent processes, which mostly resembles the Active Objects, which is a behavioral pattern for concurrent programming [182]. The Active Object pattern (also known as Concurrent Object or Actor), decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. This pattern enables more independent threads of execution, to interleave their access to data, modeled as a single object.

On the Application Layer, Shared Objects resemble the Monitor pattern, which ensures that only one method at a time executes within a passive object, regardless of the number of client processes that invoke this object's methods concurrently. The Monitor pattern is a special case



(a) Active Object pattern as Booch class diagram (Source: [182])



(b) Active Object pattern collaboration (Source: [182])

Figure 4.6: Active Object pattern

of the Active Object pattern and in general more efficient than Active Objects since they incur less context switching and data movement overhead.

Figure 4.6a depicts the Active Object pattern, running in two threads/processes: One in the client, invoking methods on the Proxy, and the other one in the Scheduler, coordination method execution.

The **Proxy** provides an interface that allows clients to invoke public methods on an Active Object. When a client invokes a method defined by the Proxy, this triggers the construction and queuing of a *Method Request* object onto the *Scheduler*'s *Activation Queue*, all of which occurs in the client's thread of control.

A **Method Request** object is used to pass context information about a specific method invocation on a *Proxy* from the *Proxy* to a *Scheduler*. A *Method Request* class defines an interface for executing methods of an Active Object. The interface also contains guard methods that can be used to determine when a *Method Request*'s synchronization constraints are met. The proxy creates instances of these classes when its methods are invoked and return any results back to clients.

An **Activation Queue** maintains a bounded buffer of pending *Method Requests*. In addition, the **Scheduler** decides which *Method Request* to de-queue next and execute on the **Servant**

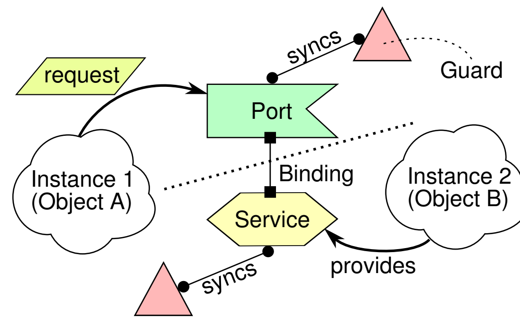


Figure 4.7: Simplified service/request scheme (Source: [119])

that defines the behavior and state hat is being modeled as an Active Object and implements all methods. This scheduling can realize any custom scheduling algorithm. A *Scheduler* typically evaluates synchronization constraints by using method request guards.

Finally, a **Future** allows a client to obtain the results of method invocations after the *Servant* finishes executing the method. When a client invokes methods through a *Proxy*, a *Future* is returned immediately to the client. The *Future* reserves space for the invoked method to store its results. When a client wants to obtain these results, it can access the *Future*, either blocking or polling until the results are computed and stored into the *Future*.

Figure 4.6b shows the three phases of collaboration in the Active Object pattern: 1. Method Request construction and scheduling, 2. Method execution and 3. Completion.

In [119] the combination of Actors and Objects has been proposed for usage in ESL for the first time. The work introduces a graph-based model for the structural representation of concurrent object-oriented systems that supports alternative behaviors as well as inheritance, polymorphism, reconfiguration and mobility. The key contribution of this work is the separation of the representation of behavior and its usage. Regarding communication, the proposed solution is based on service/request pairs, comparable to the Active Object pattern presented above.

Figure 4.7 shows the simplified service/request scheme. The Port corresponds to the Proxy, the Request to the Method Request and the Service to the Servant of the Active Object Pattern. Guards can already be applied at the Port. Handling concurrent accesses to the provided services is not further discussed.

4.3.4 CORBA- and Component-based

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) [11]. CORBA enables collaboration between networked systems, e.g. running different operating systems or using different programming languages or even different hardware. COBRA is based on the same design principles encapsulation and reuse, like object-oriented languages. If implements the distributed object paradigm [168] implemented by an Object Request Broker (ORB). An ORB is a middleware which allows program/method calls to be made from one computer to another via a computer network, providing location transparency through Remote Procedure Calls (RPC) or Remote Method Invocation (RMI), see Figure 4.8.

While having its origin in large-scale distributed and networked system, the application of ORB techniques to embedded system design has been proposed in several work.

In [157] a higher-level system communication model for object-oriented specification and the design of embedded systems based on service calls over a type-resolved dynamic network has been proposed. The approach aims lifting the abstraction of communication beyond the send-receive over a channel. It introduces a higher-level communication mechanism for system-level specification, which has features supporting object-oriented descriptions and client-server type communication modeling as in CORBA.

Figure 4.9 depicts the proposed concept of named communication (right) and compares it to explicit communication through point-to-point channels. Names communication is implemented as request-service architecture. Tasks provide Services (S) and can send Requests (R) for Services

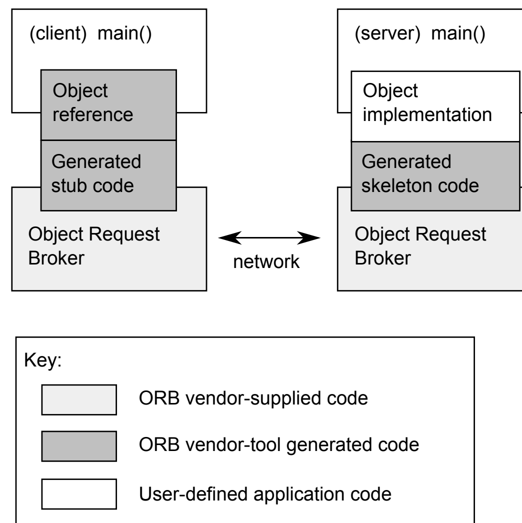


Figure 4.8: High-level paradigm for remote interprocess communications using CORBA (Source: `http://en.wikipedia.org/wiki/File:Orb.svg`)

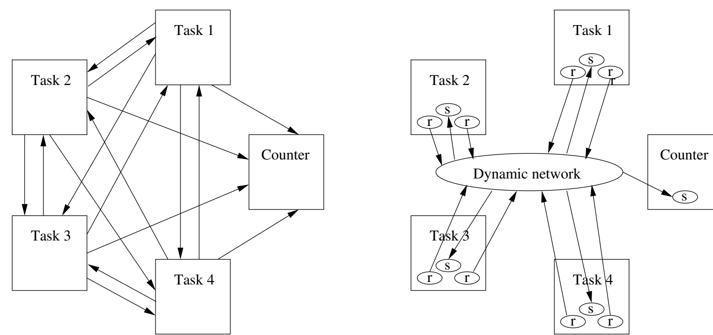


Figure 4.9: Explicit versus named communication (Source: [157])

of a dynamic network. The network is based on CORBA and dynamically routes a Request to a remote service. The work has implemented the request-service architecture as an extension to SystemC, and simulation experiments have been performed. Synthesis is not covered.

In [58, 9] an object-oriented approach to cope with the HW/SW integration problem in SoCs is presented. The proposed Object-Oriented Communication Engine (OOCE) is a system-level middleware particularly designed for SoCs which provides a high-level and homogeneous view of the system components based on the Distributed Object paradigm (see Figure 4.10). Communication between components is abstracted by means of a HW implementation of the Remote Method Invocation (RMI) semantics and all the SW and HW adapters are automatically generated from functional descriptions of the component's interface. A prototype implementation of the OOCE HW interfaces on a Xilinx ML505 has been compared with a Xilinx IPIF-based implementation [108, 112]. In the experiments, the OOCE implementation outperforms the IPIF implementation because of the unused burst capabilities in the IPIF implementation. Furthermore, this approach is focusing on the Stub and Skeleton generation. The integration and refinement of the hardware and software objects need to be performed manually. Concurrent accesses to Objects need to be resolved manually.

In [39] a component-oriented hardware/software interface specification and design methodology is presented using the CORBA Component Model (CCM). The work proposes requirement and a configurable mapping OMG IDL (Interface Definition Language) to VHDL. This work only focuses on a family of hardware interfaces enabling to represent various interaction semantics

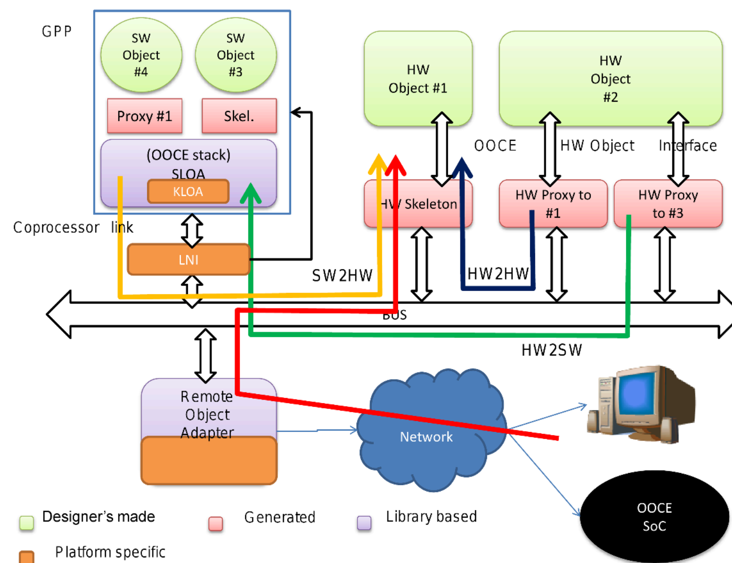


Figure 4.10: OOCE architecture (Source: [9])

and mapping configurations. In [40] an extension towards a complete semantics mapping of CORBA Interface Definition Language (IDL) and General Inter-ORB Protocol (GIOP) on the Open Core Protocol (OCP) has been investigated for hardware components and illustrated in a Software Defined Radio use-case.

In [30] a hardware/software interface design based on the concept of a Bridge Component is proposed. Bridge Components are specified in platform-specific Bridge Specification Languages (BSLs) and compiled by the BSL compilers for simulation and deployment. This is similar to the CORBA-based approaches above, but tries to be closer to the hardware domain specific languages like VHDL and Verilog. The proposed bride is basically a transactor which translates from a function call on the software side to a signal level protocol on the hardware side. The generation of RTL and TLM simulation models is discussed. Nothing is said about synthesis.

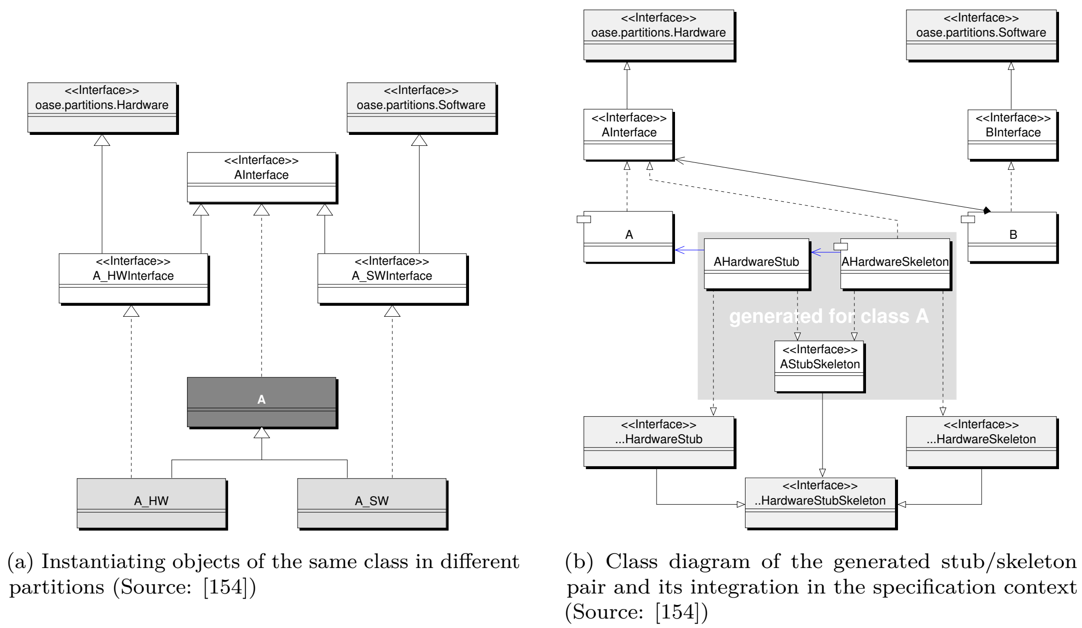
4.3.5 C++- and SystemC-based

Since this work has been implemented using C++ and the SystemC class library, an overview of other work also directly targeting an extension of SystemC for object-oriented communication will be presented in this section.

A framework for system-level partitioning of object-oriented specifications has been proposed in [154]. This work presents a strategy that allows the designer to guide the partitioning with object-oriented techniques. Furthermore, it proposes an object-oriented interface for the hardware/software partition overlapping communication of objects. Figure 4.11a shows the usage of interface base classes and multiple interface class inheritance to separate the hardware and the software view of the same class. In Figure 4.11b class `A` is realized as hardware component and a Stub-Skeleton pair for accessing `A` from the software Object `B`. The same interface class concept and Stub-Skeleton technique is used in this thesis.

In [128] the object-oriented partitioning concept of [154] been integrated into SystemC. The presented design flow includes modeling with UML, hardware/software partitioning and transformation of object-oriented specifications to regular SystemC.

In [133] an early extension of SystemC for communication refinement, called IPSIM, has been proposed. It supports an object-oriented design methodology, separates IP modules into behavior and communication components and further establishes two inter-module communication layers. The Message Box layer includes generic and system-specific communication, while the driver layer implements higher-level user-defined communications compared to a device driver. The approach mainly focuses on hardware IP integration and the integration of software driver functionality



(a) Instantiating objects of the same class in different partitions (Source: [154])

(b) Class diagram of the generated stub/skeleton pair and its integration in the specification context (Source: [154])

Figure 4.11: System-level hardware/software partitioning of object-oriented specifications

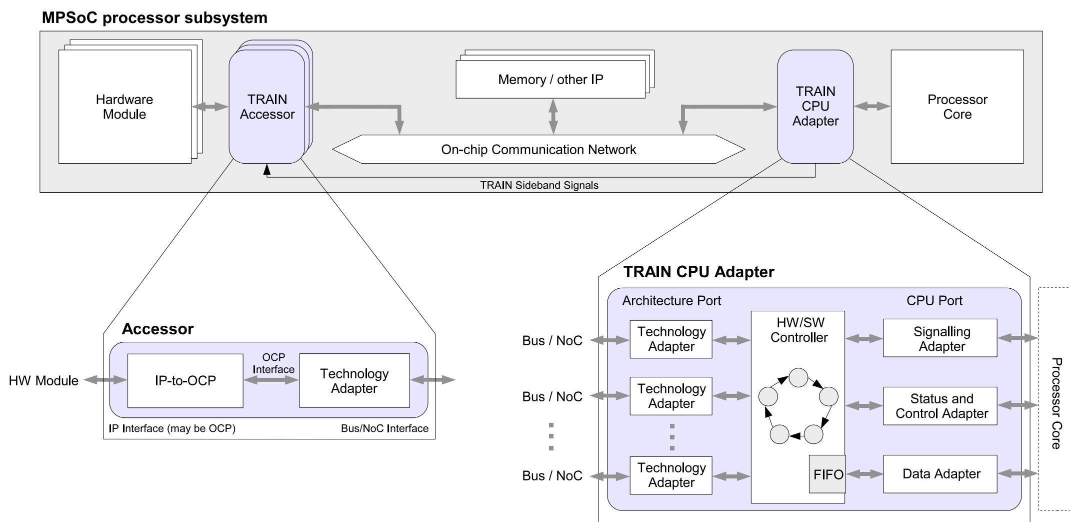


Figure 4.12: TRAIN architecture overview (Source: [81])

in a hardware/software simulation environment. It does not address hardware/software agnostic system-level specifications.

TRAIN (TRAnsaction INterchange) [81] is a Virtual Transaction Layer (VTL) architecture for TLM-based hardware/software co-design for MPSoC platforms. The proposed VTL concept allows to directly mapping transaction-level communication channels onto a multiprocessor SoC implementation. VTL is above the physical MPSoC communication architecture, acting as a hardware abstraction layer for both hardware and software components. TLM channels are represented by virtual channels, which efficiently route transactions between software and hardware entities through the on-chip communication network.

Figure 4.12 gives an overview of the TRAIN architecture. The CPU adapter routes transactions between software processes and hardware modules connected to the on-chip communication network. Accessors connect the hardware IP's interface to the target bus or NoC using a

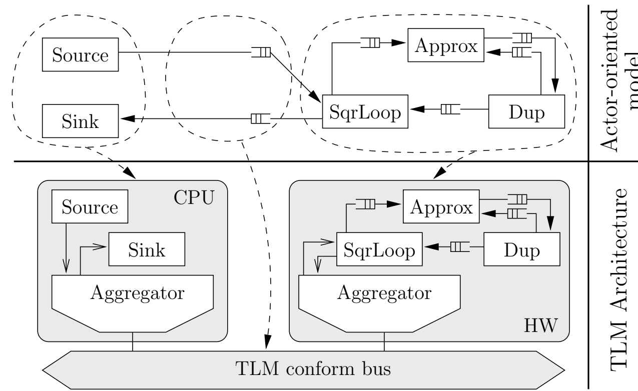


Figure 4.13: Mapping of an actor-oriented model to TLM architecture components (Source: [64])

Technology Adapter (TA). These are used to connect the CPU adapters and accessors to the platform's communication architecture.

In the high-level model, transactions between system components are described by abstract point-to-point TLM channel method calls like `put(data)` and `get(data)` supporting a generic message passing infrastructure. The presented approach does neither explicitly support an application level object-oriented techniques nor user-defined service-based communication as presented in the previous section.

In [67] the concept of [81] has been extended towards user-defined service calls. Instead of generic `read/write` or `put/get` functions, the concept of Hardware Procedure Calls (HPC) is applied. Through HPC, hardware services are provided to software processes as remote methods on top of transaction-level communication channels. This is another example for a lightweight RPC or RMI implementation focusing on the integration of hardware IPs in software dominated MPSoCs.

In [64] the mapping of actor-oriented models to TLM architectures is presented. In particular, it describes an approach to map and implement FIFO-Channel based synchronization and data transport between actors to SystemC TLM 2.0 compliant bus protocols.

Figure 4.13 provides an overview of the Actor model to TLM bus mapping. The key contribution of [64] is the proposed methodology, which enables to map actor communication in actor-oriented SystemC models (called SysteMoC [31]) to TLM architectures without any reimplementation effort. The mapping and refinement concept with minimal reimplementation effort is similar to the approach taken in this thesis.

The work in [6] aims unified design of hardware and software components using plain C++ making extensive use of template meta-programming techniques and design patterns. Moreover, aspect-oriented programming and object-oriented programming techniques have been proposed in order to provide unified C++ descriptions of embedded system components. The basic idea is to carefully isolate aspects that are specific to hardware and software scenarios. Aspects that differ significantly in each domain, such as resource allocation and communication, have been isolated in aspect programs that are applied to the unified descriptions before they are compiled to software binaries or synthesized to dedicated hardware using high-level synthesis tools. In summary, [6] provides an overview of best design practices to be applied when using plain C++ as entry for embedded hardware/software design. Communication specific aspects are only covered superficially, relating to the concept of communication encapsulation in channels and hardware/software specific proxies. Target SoC specific aspects during synthesis are not covered. These are hidden in the EPOS library.

4.3.6 Summary & Discussion

This section presented an overview of object-oriented communication concepts proposed for and used in ESL design. The combination of actors and objects is a commonly used technique

to express concurrency in classic sequential object-oriented descriptions. The Active Object is a generic pattern used to implement actors. Also OSSS, as proposed in this work makes use of this pattern. More specifically, the Application and Virtual Target Architecture Layer models use a combination of actors and objects. After mapping Shared Objects on the Virtual Target Architecture Layer, client and Shared Object are running in two independent processes, which mostly resembles the Active Objects. While on the Application Layer, Shared Objects resemble the Monitor pattern, which ensures that only one method at a time executes within a passive object, regardless of the number of client processes that invoke this object's methods concurrently. The Monitor pattern is a special case of the Active Object pattern and in general more efficient than Active Objects since they incur less context switching and data movement overhead. Since OSSS targets embedded SoCs with limited processing and communication resources, operating under tight extra-functional constraints (e.g. timing and power), some features of the Active Object pattern have not been considered in this work. Among these features are asynchronous method calls using future objects for fully decoupling method call requests from return argument handling. This interesting feature needs to be considered carefully, since it can induce non-determinism and unbounded memory demands, if not used under certain constraints. This would be subject to future work.

The Common Object Request Broker Architecture (COBRA) enables to describe the object interfaces and implemented services independently from its implementation (and implementation language), call methods on remote objects running on different nodes in a computer network, and to dynamically look-up and find remote services. Several work has been inspired by this architecture and proposed similar implementations for embedded SoCs. Also in OSSS the concept and implementation of the Remote Method Invocation (RMI) Channels stub and skeleton and marshaling/serialization concepts have been inspired by CORBA's RPC/RMI techniques. Like most related work, limitations with respect to dynamic object look-up and dynamic object and service integration and discovery apply to OSSS. Instead, only statically mapped communication links can be used with OSSS RMI Channels. Furthermore, OSSS does not support an implementation language independent interface description as proposed by several related work.

4.4 SoC communication modeling

This section gives an overview of TLM communication concepts and frameworks for SoC platform communication modeling. Many TLM layers and terminologies have been suggested by multiple authors and groups [61]. We adopt the terminology of the SystemC TLM group, but give a more precise definition of the layers based on Niemann's and Haubelt's concept of atomic transactions [69]:

Functional View (FV) models employ untimed point-to-point communication, focusing purely on functional behavior.

This layer covers the Untimed Functional Model (UFM) as introduced in Section 3.4.4.

Programmer's View (PV) models use blocking transactions and passive targets to provide a basis for early software development. Timing may be modeled coarsely, called "Programmers View with Timing" (PV+T), bus arbitration is typically not modeled.

This layer covers the Timed Functional Model (TFM) as introduced in Section 3.4.4.

Architecture View (AV) – also called cycle-approximate (CX) – models resemble the bus architecture and arbitration of an implementation platform with approximated timing. They are used for timing/performance estimation. The CCATB (Cycle-Count Accurate at The Boundaries) modeling abstraction maintains observable cycle count accuracy at the boundary of every read or write transaction occurring in the system. Since we are not concerned with maintaining accuracy at every cycle boundary, we can speed up both the simulation speed and the modeling effort. The CCATB modeling abstraction thus allows fast simulation of system models, similar to TLM while maintaining overall cycle accuracy, like in CA models which is essential for accurate system exploration.

This layer covers the Transaction Level Model (TLM) and the Computation Cycle-Accurate Model (CCAM) as introduced in Section 3.4.4.

Verification View (VV) – also called cycle-accurate (CA) or cycle-count (CC) accurate – models are clocked and represent the exact bus behavior in each cycle. Such models serve as a reference against which synthesizable RTL implementations are verified.

This layer covers the Bus Cycle-Accurate Model (BCAM) and the Cycle-Accurate Model (CAM) as introduced in Section 3.4.4.

The following overview of TLM communication concepts and frameworks is based on the analysis performed in [82, 49].

4.4.1 SystemC TLM

Founded in 2003, the OSCI TLM Working Group (TLM-WG) pioneered the development of TLM frameworks for SystemC with the release of the OSCI TLM kit 1.0 [114]. A set of three interfaces is provided that form the heart of the kit, enabling unidirectional blocking, unidirectional non-blocking, and bidirectional blocking transfers.

The TLM 1.0 kit does not define standard data types nor abstraction levels. The intended use of the kit is to develop customized channels with it, using application-specific data structures and user-defined protocols. Thus, the OSCI kit does not help in achieving model interoperability. Having identified these issues, the OSCI TLM Working Group started working on a revised version of the kit (OSCI TLM 2.0). A first proposal of concepts has been made available for public review in spring 2007 [68].

Since 2012, TLM 1.0 and TLM 2.0 have become part of the IEEE Std 1666-2011 (IEEE Standard for Standard SystemC Language Reference Manual) [13], which is the first revision of the initial IEEE Std 1666-2005.

4.4.2 GreenBus

The GreenBus [82, 49] is a generic bus model that introduces the concept of transactions consisting of uninterruptible phases (atoms) which are collections of payload values (quarks). By identifying bus protocol signals as quarks, the generic model can be customized to model concrete buses. Simulation can be performed at the layer of transactions, atoms or quarks, corresponding to approximately timed, cycle-approximate, and cycle-accurate layers.

The GreenBus concepts have been submitted to the Open SystemC Initiative (OSCI) as interoperability standard proposal, and have been partially adopted in the SystemC TLM 2.0 standard. GreenBus offers a compatibility layer for SystemC TLM 2.0 through sockets that convert the TLM 2.0 Generic Payload into the corresponding GreenBus data transaction container and vice versa.

4.4.3 Accuracy-Adaptive TLMs

Transaction level models exist at different abstraction levels, which are characterized as untimed, approximately timed, cycle-approximate and cycle-accurate. These abstractions provide different trade-offs between abstraction and accuracy suitable for different use models. The decision, which abstraction level to employ is usually made by the user prior to simulation based on the use case and required accuracy. In [54] a modeling technique that allows covering several abstraction levels in a single model and switching between these abstraction levels at any time, in particular dynamically during simulation. This feature is employed to automatically adapt simulation accuracy to an appropriate level depending on the model's state, leading to an improved trade-off between simulation performance and accuracy. In [32] this adaptive TLM modeling has been successfully applied to the AMBA AHB bus and the Fast Simplex Link (FSL) based on SystemC TLM 2.0. In [15] a modeling style for systematic construction of accuracy-adaptive TLM models is presented.

4.4.4 OCP SystemC channels

The OCP-IP [86] provides a comprehensive library of point-to-point channels for modeling of SoCs based on the Open Core Protocol with SystemC [70]. Three levels of abstraction are supported, namely OCP-tl0 for cycle accurate, OCP-tl1 and OCP-tl2 for CCATB, and OCP-tl3 for PV communication modeling. A large set of interface methods for both blocking and non-blocking channel access is provided. OCP channels use predefined data types for transfer qualifiers such as address, command, thread identifier, etc. and provide basic instrumentation for transaction monitoring. However, they do not provide any means for communication architecture simulation, as they only support point-to-point communication.

4.4.5 STMicroelectronics TAC

TAC (Transaction Accurate Communication) [117] is a protocol functional verification and embedded software development through transaction-accurate communications. It provides a set of channels and interfaces aiming at the creation of virtual PV prototypes for early software development.

TAC v1 has been proposed as TLM 1.0 standard to OSCI and TAC v2 has been aligned to the final TLM 1.0 OSCI standard. It has been built on top of OSCI TLM 1.0 and provides a pair of so-called initiator and target ports. These ports implement simple blocking read and write communication methods. As part of TAC a router model is delivered that can add wait calls to transactions so that approximate (PV+T) communication times are estimated, e.g. using weighted randomization. However, at this time, TAC did not support communication architecture simulation.

With TAC v3, the OSCI TLM 2.0 standard is supported as well and enables communication architecture simulation with register map support (which has not been standardized yet). It enables register decoding, register and bit field access control, setters & getters, register meta data (synchronization points) and a unified reporting mechanism.

4.4.6 SystemC^{SV}

The SystemC ^{SV} [155] extension of SystemC is an approach to TLM communication modeling at mixed levels of abstraction. It defines an interface that describes communication among PEs at different levels of abstraction in terms of interface items. An interface item may represent either a complete transaction, a frame or word, a field within this frame, or a signal state on a physical wire.

SystemC ^{SV} provides C++ macros with which the composition of interface items can be described in a declarative style. Interfaces use these macros to automatically decompose interface items for transmission and reassemble them on reception. Thus, mixed multi-level communication of PEs at different abstraction levels becomes possible.

While the SystemC ^{SV} approach is very appealing for describing point-to-point communication, it does not support modeling of shared-buses. Also, the simulation speed-up of 160x (as reported in [155]) when using abstract interface items instead of RTL level interface items is considerably lower than the performance reached by other TLM frameworks (e.g. [68, 82]).

4.4.7 OCCN

OCCN (On-Chip Communication Network) [121] addresses network-on-chip modeling with SystemC at mixed abstraction levels. It transports "protocol data units" (PDUs) among PEs. PDUs are composed of user-definable fields that carry user data and protocol information. An OCCN channel implements blocking send and receive methods for PDUs. Receive delays and timeouts can be specified as parameters, enabling PV+T modeling equal to ST TAC.

Coppola et al. propose the implementation of an application API on top of the PDU transport layer. This API can provide master and slave ports with convenience methods. Similar to OCP channels, the OCCN channel is a point-to-point channel that does not support communication architecture simulation.

4.4.8 IBM CoreConnect models

IBM provides SystemC bus functional models for their CoreConnect architecture [98]. They enable precise CC simulation of the PLB [57], OPB [151], and DCR [76] buses. For each of the three buses a dedicated API is provided, which are quite comprehensive and low-level. Thus, the IBM library is a good tool to create a virtual prototype of an already existing CoreConnect SoC, but is inappropriate for architecture exploration.

4.4.9 ARM AMBA models

A library of AMBA [167] bus functional models is provided by ARM. These models enable simulation of AHB [83] and AXI [123] on-chip buses. In contrast to the IBM CoreConnect models, the ARM AMBA bus functional models have been implemented on top of SystemC TLM 2.0 to integrate with existing environments and models and to foster architecture exploration.

4.4.10 CCATB AMBA models

In their paper on the CCATB simulation approach [126], Pasricha et al. present CCATB Communication Architecture Functional Models (CAFMs) for the AMBA AHB and AXI buses and compare them with ARM's cycle accurate bus functional models. A speedup of 55% is achieved, which is rather unsatisfying in comparison to the other approaches. This limited result may be due to the utilized instruction set simulator.

4.4.11 ROM

Result Oriented Modeling (ROM) [90] takes advantage of the limited observability within a transaction to increase the performance. The idea is to calculate the transmission time of a transaction a priori and then advance simulation time to the end time of the transaction. This is similar to PV+T approaches such as TAC, but in addition, ROM checks whether a "disturbing influence" has occurred, such as an overlapping transaction on a shared bus. In this case, corrective measures are taken if necessary.

Schirner et al. achieve simulation speeds close to that of PV models with this approach. However, ROM simulation results are only correct when observed at transaction boundaries. Thus, Bus Accurate (BA) and Cycle-Count accurate (CC) models are not supported. Moreover, ROM CAFMs are only periodically synchronous to SystemC simulation time.

4.4.12 NoC channels

In [137], Kogel et al. present a a combined bus/NoC simulation framework for packet based communication via a so-called NoC channel. Multiple masters and slaves can be connected to the channel. It implements a symmetric request-response communication scheme. Masters initially not compliant to this interface can be attached by means of adapters.

An advantage of Kogel's approach over all of the above proposals is that protocol simulation and the channel inherently supports multiple access arbitration. So-called network engines can be attached to the channel that implement the behavior of a bus or NoC protocol. Upon initiation of a transaction, the channel creates a transaction data structure and forwards it to the network engine. The network engine collects concurrent transaction requests in a queue and delays them in accordance with the arbitration policy. Transmission times are taken into account as well, however on a coarse-grained packet basis only. The framework therefore lacks the possibility of providing CCATB timing estimations. The authors do not consider communication refinement towards the implementation model.

4.4.13 Summary & Discussion

Table 4.1 provides an overview of the presented techniques recently used for SoC communication modeling and simulation.

Transaction level modeling as described in [13, 82, 49, 54, 15] is motivated by the need of an early platform for software development, fast system level design exploration [90] and

	Simulation Accuracy			Access Semantics		Transfer Types		Data Types		Topology		
	PV	AV	VV	B	NB	word	burst	fixed	user	p2p	bus	NoC
SystemC TLM 2.0 [13]	x	x	x	x	x	x	x		x	x	x	
GreeBus [82, 49]	x	x	x	x	x	x	x		x		x	
Accuracy-Adapt. [54, 15]	adaptive			x	x	x	x		x		x	
ROM [90]	x					x	x	x			x	
OCP TL1 [86]			x	x	x	x	x		x	x		
OCP TL2 [86]		x		x	x	x	x		x	x		
OCP TL3 [86]	x			x	x	x	x		x	x		
TAC [117]	x	x		x	x	x	x		x	x	x	x
SystemC-SV [155]	x	x		x		x	x		x	x		
OCCN [121]	x	x		x		packet			x			x
IBM CoreConnect [98]			x	x		x	x	x			x	
NoC channels [137]	x	x		x	x	packet		x	x	x	x	x
OSSS RMI Channels [22]	x			x		packet			x	x	x	
OSSS Channels [77, 78]			x	x		x	x	x		x	x	

PV	Programmer's View	B	Blocking transfer	p2p	point-to-point
AV	Architecture View	NB	Non-Blocking transfer	bus	shared bus or cross-bar
VV	Verification View			NoC	Network-on-Chip

Table 4.1: Overview of communication in SoC platforms (extended from [49])

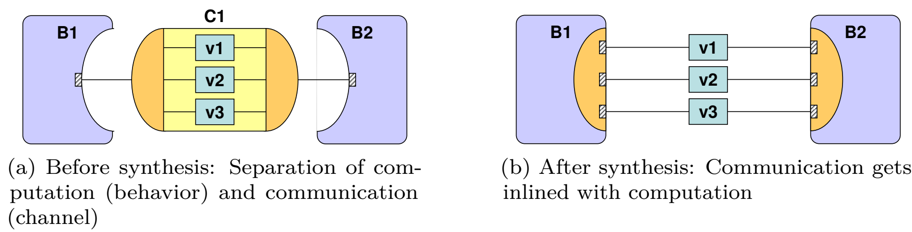
verification and the usage of system level models in block level verification. These approaches are focused on the efficient simulation and do not deal explicitly with synthesis or mapping to a physical target platform. The Open Core Protocol International Partnership (OCP-IP) tries to overcome this limitation of OSCI TLM. OCP-IP defines a bus-independent open license protocol [86] with point-to-point interfaces to a bus wrapper interface module. Since OCP does not define Layer-0 (RTL-Layer), it is not synthesizable without manual refinement.

The goal of OSSS is to enable HW/SW communication modeling with synthesizable channels. OSSS supports a two-layered channel model. OSSS RMI Channels [22] implement a communication channel transparent Remote Method Invocation protocol. One layer below OSSS Channels [77, 78] are using a set of `sc_signals` internally, which is equivalent to RTL wires and therefore synthesizable. For shared bus and point-to-point channel modeling a predefined read/write interface is proposed. A protocol library contains different implementations of these interface methods, e.g. to simulate the OPB. Data transport and arbitration in OSSS is performed at RTL abstraction level. Thus, bus simulation is cycle accurate. As a drawback, the simulation performance achieved with this approach is hardly better than with pure RTL models. Instead of using synthesizable OSSS Channels, our RMI Channels can also be used on top of SystemC TLM channels (see [48, 23, 17]) which is not covered in this thesis.

4.5 SoC Communication Synthesis

Starting from a strict separation of computation and communication (see Section 3.6.2), this work targets SoC communication synthesis as a synonym for channel and interface synthesis. A channel encapsulates the entire communication protocol and may be hierarchically refined to represent the physical communication structure (i.e. signals or wires). During channel synthesis, the protocol state-machines at the channel interface implementations are realized as part of the computation, as either software of hardware implementation (as shown in Figure 4.14).

As defined in [79]: “It is important to note, that we strictly differentiate between the terms interface adapter and interface, even though the term interface is often used as synonym for interface adapter in the literature. An interface is an integral part of a communication component while an interface adapter is a self-containing entity that interconnects interfaces. Due to the ambiguity of the term interface, existing interface synthesis approaches cover the construction

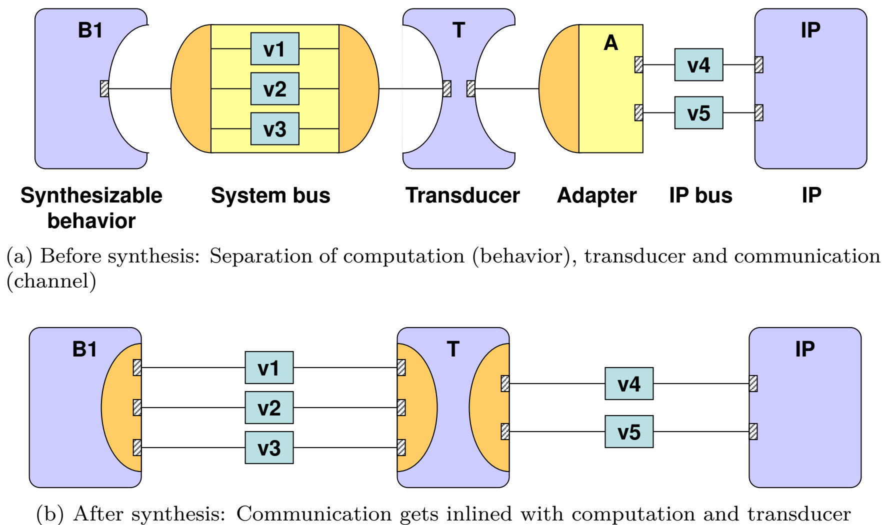


(a) Before synthesis: Separation of computation (behavior) and communication (channel)

(b) After synthesis: Communication gets inlined with computation

Figure 4.14: Channel synthesis in SLDLs [160]

of component interfaces as well as the synthesis of self-containing protocol adapters."



(a) Before synthesis: Separation of computation (behavior), transducer and communication (channel)

(b) After synthesis: Communication gets inlined with computation and transducer

Figure 4.15: Channel synthesis with transducer in SLDLs [160]

A transducer, as shown in Figure 4.15, performs protocol translation between two different protocols. A transducer can be used to connect two different channels, implementing different protocols. Furthermore, it can be used to connect IP components with a different physical interface and logical protocol. In this case, an adapter translates the protocol to the physical interface and the transducer translates between the adapter and the other protocols.

The work in [79] itself combines interface synthesis techniques with reconfigurable computing concepts to create an interface adapter module (called Interface Block (IFB)), which affords the deterministic reconfiguration of tasks at runtime. Furthermore, [79] presents an integrated design flow which supports the modeling and automated synthesis of an IFB. Using worst-case-execution-time analysis and schedulability analysis, real-time protocol conversion capabilities of an IFB can be proven. With respect to its reconfigurablity is goes beyond the scope of interface synthesis covered in this thesis where interfaces and the number of communication partners are statically fixed during design time. The concept of IFB could be applied as an alternative to the currently used Xilinx IPIF adapter (see Section 7.8).

Concerning communication synthesis from abstract channel and protocol specifications to RTL, several works exist in literature. We do not intend to give an exhaustive overview here, but rather focus on work applicable to the presented channel, transducer and adapter concept as used in SLDLs such as SpecC and SystemC.

4.5.1 SpecC-based

SpecC [160] is completely refinement driven and introduces lots of useful concepts to perform a separation of computation and communication. It uses hierarchical behaviors for the specification of computation and channels for the specification of communication.

The protocol translation can be implemented by user-defined adapter modules. These wrapper/adapter modules have to be written in the SpecC language and become part of the SpecC synthesis flow [116]. An automated layer-based generation of System-on-Chip bus communication models is presented in [63].

Another approach [189] proposes to reduce protocol specifications to the combination of five basic operations (data read/write, control read/write, time delay). The protocol description is then broken into blocks (called relations) whose execution is guarded by a condition on one of the control wires or by a time delay. Finally, the relations of the two protocols are matched into sets that transfer the same amount of data. Although this algorithm is able to account for data width mismatch between the two modules, the procedural specification of the protocols makes it difficult to adapt different data sequencing, so that only the synchronization problem is solved. The concept was extended to IP based design in [115] and an automatic communication refinement for system level design in [129].

A different work is that of Boriello [192], who introduces the "event graph" to establish the correct synchronization of data sequencing. The limitation of this approach is that the two protocols should be made compatible by manually assigning labels to the data on both sides, since the specification of the protocols is given at a very low-level of abstraction using waveforms.

In [177] Interface Based design was proposed as a methodology that attempts to orthogonalize the communication and the behavior of IPs. Therefore, IP blocks are abstracted to a transition or messaging level. In this abstract view, complex blocks exchange messages using a robust user-defined type system rather than using complex signaling conventions based on conventional wires. The abstracted communication leads to an improvement in the simulation performance. However, this abstract level is not sufficient for the automated implementation of the protocol translation. One solution to get to the final implementation is the stepwise refinement of the protocols down to a pin- and cycle-accurate bus functional model.

4.5.2 SystemC-based

For constructing communication in system level models, SystemC™ [13] has adopted the idea to orthogonalize communication and behavior. Therefore it has introduced the concept of primitive and hierarchical channels. Our proposed approach for SystemC channel synthesis is to allow user-defined abstract synchronization and communication on transaction or message level granularity. The synchronization behavior of the channel is preserved during synthesis, while the transaction-based communication is automatically translated to a pin- and cycle accurate communication channel.

Müller and Siegmund presented a SystemC based specification for protocols [155]. From this specification a deterministic interface adapter FSM is generated [147, 146]. In order to avoid a consideration of the complete product FSM, an architecture template is introduced. For the resulting interface adapter FSM, VHDL code is generated as input for a low-level synthesis. In our approach we also enable the use of IP communication resources through the generation of interface protocol FSMs, e.g. for shared busses. A synthesizable specification of the interface protocol FSM or the entire communication network has been presented in [77].

4.5.3 Commercial SystemC and C/C++ synthesis tools

This section briefly discusses the communication and interface synthesis capabilities of a selection of commercial SystemC and C to RTL synthesis tools.

4.5.3.1 SystemCrafter

SystemCrafter [230] is a SystemC to VHDL and Verilog synthesis tool. It also support SystemC output to compare the synthesized design with the input model. The SystemCrafter user manual [231] language reference reveals that this tool only partially supports the SystemC synthesizable subset [33]. SystemCrafter does not support user-defined classes, structs and unions, multi-dimensional arrays typedefs and templates. Advanced control statements like

continue and break are also not supported. From a modeling perspective these are severe limitations.

Regarding communication, SystemC modules can only communication using sc_signal. No other SystemC channels are supported. Thus when using user-defined channels, these need to be manually refined at RT level using ports and signals.

SystemCrafter supports predefined RAM modules sc_module_ram_craft including support for Xilinx Block RAMS.

4.5.3.2 Handel-C

Handel-C [213, 214] is a subset of C, with non-standard extensions to control hardware instantiation with an emphasis on parallelism. Parallel behavior is described using Communication Sequential Processes (CSP) [198] keywords (like seq and par in Occam). As defined in CPS, channels provide communication between parallel threads. These channels are directed: i.e. one of the paths outputs data onto the channel and the other parallel thread can read the data. Channels can be created with or without a FIFO capability. For a channel without a FIFO the first thread to execute the channel read (or write) command waits until the corresponding write (or read) is executed in the other communicating thread. In this way, the sender and receiver are performing a rendezvous where they can pass data and synchronize their operation in a cooperative manner. Handle-C is mainly targeted for FPGA platforms.

4.5.3.3 C-to-Silicon

Cadence C-to-Silicon compiler [208] supports large parts of the SystemC synthesizable subset [33] and the transaction-level modeling TLM 1.0 API [93]. With this approach, the designer is capable of building custom point-to-point interfaces. As a high-level synthesis tool, it capable to automatically schedule SC_THRERADs but also works with manually scheduled SC_CTHREADs. These features enable a manual protocol refinement down to pin level accuracy when necessary.

Regarding more advanced communication it has built-in support for flex channel block-to-block streaming communication and enables to interface AXI3 and AXI4-Lite on-chip interconnect fabrics.

For modeling and integration of memories, different approaches are available. C-to-Silicon is capable to automatically flatten multi-dimensional arrays into a flat memory array, supports platform specific built-in RAM (e.g. Xilinx Block RAM).

With these features, C-to-Silicon provides support for manual refinement and interface inlining for user-defined channels and limited automatic synthesis support for pre-defined point-to-point connections and AXI on-chip interfaces. This does not explicitly support the designer in exploiting knowledge about communication and synchronization properties from the algorithmic description, but requires a manual mapping to the available communication mechanisms.

4.5.3.4 Cynthesizer

Forte Design Systems Cynthesizer [211] fully supports the SystemC synthesizable subset [33] and allows use of sc_port for abstract communication between modules and to automatically switch between OSCI TLM 1.0 and pin-level communication interfaces. The provided technique supports integration of Cynthesizer with any OSCI TLM 1.0 environment for increasing the simulation performance. To support this TLM feature for synthesis, TLM versions of behavioral IP for FIFOS, memory interfaces, and streaming interfaces are provided as both, high-speed TLM simulation and pin-accurate synthesizable models. The Cynthesizer approach also uses SystemC primitive channel ports for the specification of user-defined communication. In contrast to our approach, only a transactor synthesis to convert from a method-based interface to a pin-level interface is provided. The benefit of our approach is that we also consider information from the high-level channel model with its user-defined communication and synchronization properties and perform automatic protocol synthesis.

4.5.3.5 CatapultC

Calypto's Catapult [209] does not require the definition of explicit interface protocols when starting with a pure ANSI C++ description. But Catapult also supports large parts of the SystemC synthesizable subset [33]. The usage of SystemC instead of C++ enables to take advantage of the SystemC modular hierarchy and data types.

Catapult is a true high-level synthesis tool. During behavioral synthesis, interfaces between behavioral blocks are mapped to either streaming, single- or dual-port RAM, handshaking, FIFO, or custom built-in I/O components. For communication intensive designs with tight constraints or the integration with complex bus interfaces, Catapult also allows designers to specify their interface requirements using cycle-accurate descriptions in SystemC (e.g. `SC_CTHREAD`s with `wait` statements at clock boundaries). For this purpose, CatapultC provides its own transaction-level modeling style, similar to OSCI TLM 1.0. This is basically the same approach with the same limitations as used in the Forte Cynthesizer described above.

4.5.3.6 Vivado

The Xilinx High-Level Synthesis software Vivado HLS [235, 3] (former XPilot from AutoESL [35]) transforms a C, C++ or SystemC specification into a Register Transfer Level (RTL) implementation that synthesizes into a Xilinx FPGA. Vivado supports abstraction of algorithmic description using C and C++, data type specification (including integer (arbitrary precision), fixed-point or floating-point) and communication interfaces (FIFO, AXI4, AXI4-Lite, AXI4-Stream). Optimized for Xilinx FPGAs, it automatically makes use of Xilinx on-chip memories (distributed RAM and Block RAM) and DSP elements.

Interface synthesis in Vivado is handles differently for the C/C++ and the SystemC entry. In general, Vivado HLS does not perform interface synthesis on SystemC. Communication between threads and methods (wrapped by modules) should only be performed using `sc_buffer` or `sc_signal` channels. Furthermore, Vivado supports interface synthesis for some memory interfaces, such as RAM (`ap_mem_if`) and FIFO (`sc_fifo_in`, `sc_fifo_out`) ports. For SystemC Vivado HLS directives to partition the array ports into individual elements are provided. The Vivado high-level synthesis process may add additional clock cycles to a SystemC design to meet timing requirements.

The main SystemC limitations in Vivado are:

- An `SC_MODULE` cannot be nested inside another `SC_MODULE`. Forcing the designer to manually flatten the design hierarchy before synthesis.
- An `SC_MODULE` cannot be derived from another `SC_MODULE`. Thus effectively preventing inheritance anomaly for SystemC designs.
- No support for `SC_THREAD` processes; only clocked threads (`SC_CTHREAD`) are supported. This restricts SystemC to behavioral RTL design only.

The C/C++ entry provides more high-level synthesis features, but less control on the resulting structural representation of the synthesis result, compared to the SystemC entry. Regarding interface and protocol synthesis, the following is supported:

- Interface types `ap_ctrl_none`, `ap_ctrl_hs` and `ap_ctrl_chain` are used to specify if the RTL is implemented with block-level handshake signals or not. Block-level handshake signals specify when the design can start to perform its standard operation and when that operation ends. These interface types are specified on the function or the function return.
- Array arguments are typically implemented using the `ap_memory` interface. This type of port interface is used to communicate with memory elements (RAMs, ROMs) when the implementation requires random accesses to the memory address locations. Array arguments are the only arguments that support a random access memory interface. If only sequential access to the memory elements is required, the `ap_fifo` interface should be used.

- An `ap_bus` interface can communicate with a bus bridge. The interface does not adhere to any specific bus standard but is generic enough to be used with a bus bridge that in-turn arbitrates with the system bus. The bus bridge must be able to cache all burst writes. An `ap_bus` interface can be used in two ways: *standard mode* of operation is to perform individual read and write operations, specifying the address of each and in *burst mode*, the base address and the size of the transfer is indicated by the interface (i.e. the data samples are transferred in consecutive cycles).

- An AXI Stream I/O protocol can be specified as the I/O protocol using mode `axis`. An AXI Slave Lite I/O protocol can be specified as one of the I/O protocol using mode `s_axilite`. An AXI Master I/O protocol can be specified as one of the I/O protocols using mode `m_axi`. A complete description of all AXI interfaces including timing and ports can be found in [2].

4.5.3.7 eXCite

eXCite [210, 47] is an ISO/ANSI-C (including pointers, structs and type definitions) to Verilog or VHDL RTL high-level synthesizes tool. The resulting VHDL or Verilog code is suitable input to FPGA (Altera, Xilinx, Actel) or ASIC logic synthesis tools. eXCite has built-in data for devices from Altera, Xilinx and Actel. A technology library generator is also available to customize ASIC library information.

For communication, channels that specify how the generated hardware block will communicate with its surroundings can be inserted. These channels can be streaming, blocking, or indexed, like arrays. These channels are accessed through simple C procedure calls in the C code or by specifying pragmas [206]. A channel library for an Altera FPGA platform with Avalon streaming and memory-mapped interfaces is included. Other platform's standard memory, handshake, and streaming interfaces can be manually integrated.

4.5.4 Summary & Discussion

In this section, an overview of related work in communication channel and interface synthesis has been provided. It has been mainly focused on synthesis from C- and C++-based SLDL, including some state-of-the-art synthesis tools.

In this thesis, an alternative concept for SystemC channels is proposed. Shared Objects, like any other objects may specify a set of methods that form its interface. This allows processes to communicate and exchange data via Shared Objects using the Interface Method Call (IMC) mechanism similar to SystemC channels.

The presented related work follows two basic principles for interface and channel synthesis: custom refinement and mapping to pre-existing communication adapters. In this work, a combination of both principles will be applied. The Shared Object will be synthesized to custom hardware, following the basic idea of [135, 97]. For the connection of clients to a Shared Object a library-based communication adapter synthesis will be applied. In the resulting communication architecture, the Shared Object can be regarded as a Transducer or Bridge that performs protocol conversion and guarantees data consistency among its connected client processes.

Compared to arbitrary channels, Shared Objects have a clear synthesis semantics, possess built-in mechanisms for handling concurrent accesses, and exhibit a timed behavior during simulation. The handling of concurrent accesses is basically realized by means of a scheduler that can be specified by the user for each Shared Object. The scheduler determines which client process is granted to access the Shared Object in the case of concurrent requests. All other requesting clients are blocked meanwhile. Consequently, accesses are mutual exclusive, which is supported by a guard mechanism. Additionally, the timed behavior of Shared Objects give the designer an early and realistic impression on the temporal behavior of the modeled system during simulation even before performing synthesis.

This work is based on the explicit exploitation of object-oriented constructs used in hardware design as proposed by [162, 128]. Grimpe et. al. [135, 97] have introduced the concept of Guarded/Shared and Polymorphic Objects to further raise the abstraction of object-oriented hardware descriptions while keeping its synthesizability in mind. However, the communication

between the client processes and the Shared Object followed a pre-defined and static point-to-point communication protocol only, and did not follow the SystemC channel port-interface binding concept. In [22] an evolution of the Shared Object directly following the SystemC port to interface binding and IMC communication style has been presented. This is much closer to the SystemC channel description and allows much easier replacement of SystemC primitive channels with Shared Objects. Furthermore, this provides a hook for selectively refinement and synthesis to different user-defined cycle accurate communication protocols.

Existing commercial C, C++ and SystemC synthesis tools, e.g. [211, 209, 208, 235] allow behavior level synthesis. However, these tools are well suited for synthesis of complex functional blocks, but lack adequate support for flexible and predictable communication synthesis[5] between these functional blocks. Some tools offer pre-defined point-to-point communication resources like FIFO buffers (supporting TLM1.0 interfaces), RAM or dedicated bus interfaces for synthesis. At the beginning of this work most of these communication and interface synthesis support has not been available in commercial tools. For this reason the *Fossy* SystemC synthesis tool [225] has been developed in the three successive EU projects ODETTE, ICODES and ANDRES. With the recently available Vivado HLS tool [235], a large part of the complexity of *Fossy* could be moved to Vivado and thus only generating synthesizable SystemC code for Shared Objects and client interfaces.

This work proposes a new SystemC-based coding style for high-level synthesis using Shared Objects – that allows deterministic access scheduling and user-constrained refinement – for HW/HW communication, which shall make it easier to convert communication parts from abstract SystemC description to synthesizable RTL.

4.6 Electronic System-Level Synthesis Methodologies

For a high-level comparison of the overall presented OSSS methodology, this section gives an overview of major ESL system-level synthesis methodologies. It has been extracted from [28]. For more details on these methodologies, please refer to the original source.

4.6.1 Daedalus

“Daedalus provides an integrated and [...] automated framework for system-level architectural exploration, system-level synthesis, programming, and prototyping of heterogeneous Multi-Processor System-on-a-Chip (MPSoC) platforms [72], [53]. The Daedalus design flow [...], leads the designer in a number of steps from a sequential application (i.e., behavioral specification) to an MPSoC system implementation on an FPGA with a parallelized version of the application mapped onto it. This means that Daedalus includes or interfaces with component- and task-level back-end synthesis processes to produce an MPSoC implementation at the RTL and ISA levels for hardware components and software processes, respectively. Since the entire design trajectory can be traversed in only a matter of hours, it offers great potentials for quickly experimenting with different MPSoCs and exploring a variety of design options during the early stages of design.

[...]

[The Daedalus framework has been mainly designed for dataflow dominated applications from the multimedia, imaging, and signal processing domains], that naturally contain tasks communicating via streams of data. Such applications are conveniently modeled by means of the Kahn Process Network (KPN) MoC [200]. The KPN MoC [...] is a dataflow network of concurrent processes that communicate data in a point-to-point fashion over bounded FIFO channels, using blocking read/write on an empty/full FIFO as synchronization mechanism. The KPNs that Daedalus operates upon can be manually derived or automatically generated. In the latter case, behavioral input specifications are sequential C programs. [...] To allow for

[5]Predictable in the sense that the resulting communication timing is reflected in the input simulation model. Usually, the effect of communication synthesis can only be observed on the generated output model of the synthesis tool. For this purpose, most synthesis tool offer a SystemC output model that can be used with the original testbench.

automatic translation into a KPN, these C applications need to be specified as so called Static Affine Nested Loop Programs (SANLPs) [73] [...].

In terms of target MoA, Daedalus considers MPSoC platforms in which both programmable processors and dedicated hardwired IP cores are used as processing components. They communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication and synchronization between processors are realized by blocking read and write primitives. Such platforms match and support the KPN operational semantics very well, thereby achieving high performance when KPNs are executed on the platforms. Also, directly supporting the operational semantics of a KPN, i.e., the blocking mechanism, in the target platforms allows the processors to be self-scheduled. This means that there is no need for a global scheduler in the platforms.

Daedalus architectures are constructed from a library of predefined and pre-verified IP components. These components include a variety of programmable processors, dedicated hardwired IP cores, memories, and interconnects, thereby allowing the implementation of a wide range of heterogeneous MPSoC platforms. [...] Daedalus produces platforms in the form of synthesizable VHDL (i.e., a netlist MoS) together with the C code for KPN processes that are mapped onto programmable processors. [...] Daedalus designs can be readily implemented on an FPGA for prototyping.

Daedalus supports the mapping of multiple KPN processes onto a single processor, [either by performing compile-time scheduling or by using a lightweight multi-threading operating system].

The Daedalus design process is guided by automated DSE, which uses a MoP that combines a [Task Accurate Performance Model (TAPM)] and an [Instruction Set Accurate Performance Model (ISAPM)] to evaluate design instances. Moreover, [the Daedalus] computation synthesis [flow] is fully automated, while its communication synthesis is semi-automatic as it uses communication IP components which may need to be customized by hand.

[...]

[The Daedalus] design flow consists of three key steps, which are implemented by the *KPNgen*, *Sesame* and *ESPAM* tools respectively.

- *KPNgen* [73] allows for automatically converting a sequential (SANLP) behavioral specification written in C, into a concurrent KPN [200] specification. By means of automated source-level transformations, KPNgen is also capable of producing different input-output equivalent KPNs, in which for example the amount of concurrency can be varied. Such transformations enable behavioral-level DSE.
- The generated or handcrafted KPNs are subsequently used by the *Sesame* modeling and simulation environment [88] to perform system-level architectural DSE. [...] Sesame uses (high-level) architecture model components from the Daedalus IP component library [...]. Sesame allows for evaluating the performance of different design decisions in terms of target platform architectures (i.e., resource allocation), binding of KPN processes to architecture resources, and scheduling policies. [Sesame supports fast TAPM-level simulations and a gradual refinement of its architecture performance models down an ISAPM-level simulation model to increase accuracy.] [...] Besides exhaustive simulative DSE [...], Sesame also supports heuristic search methods, such as genetic algorithms, to steer DSE in larger design spaces. Moreover, it includes an additional design space pruning step, which is based on analytical models and takes place before DSE to trim the design space that needs to be studied using simulation.
- Sesame's DSE results in a set of promising system design candidates, each of which are described using a XML-based platform description [...] and process binding description. [These] descriptions, together with the (behavioral) KPN description, act as input to the *ESPAM* tool [52]. This tool subsequently uses RTL versions of the components from the IP library to automatically generate synthesizable VHDL that implements the candidate MPSoC platform architecture. In addition, it also generates the C code for those KPN processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can be [implemented] onto an FPGA for prototyping.

[...]" ([28] © 2009 IEEE)

4.6.2 System-On-Chip Environment

"The System-On-Chip Environment (SCE) realizes an interactive and automated design flow with a consistent and seamless tool chain all the way from specification down to hardware/software implementation [...] [37]. Starting from an abstract, behavioral specification of the desired system functionality, the SCE ESL synthesis frontend allows for interactive, user-driven exploration of the system-level design space. Given design decisions and database components, SCE will automatically implement the specification on the given target platform and in the process generate structural TLMs of the system architecture at various levels of abstraction. In a component- and task-level backend process, hardware and software processors in the TLMs are then individually synthesized further down to their final RTL and ISA implementations, respectively.

SCE is based on the SpecC SLDL and methodology [160]. [...]

At the input of the SCE [...] design flow, the behavioral system-level specification provides the designer with an abstract, high-level model for parallel programming of the platform across hardware and software processors. Computation is specified in a hierarchical and concurrent fashion following a Program State Machine (PSM) MoC [188]. [see Section 3.6.2 for PSM definition] [...]

ESL refinement tools will then take an input specification and automatically implement it on a given target platform based on a given mapping. Through its processing element (PE), communication element (CE) and bus databases, SCE supports a system-level MoA that allows for heterogeneous, bus-based MPSoCs consisting of PEs, such as custom hardware and programmable software processors, IP blocks, and memories, connected through complex networks of buses and CEs, such as bridges and transducers.

At the output of the ESL design front-end, intermediate TLMs represent a system-level MoS that serves as a virtual prototype of the application computation and communication running on the platform processors, memories and buses. System TLMs automatically generated by SCE integrate high-level, task-accurate MoPs (TAPMs) with back-annotated task code running on top of abstract OS and processor models to provide analysis and design validation without the need for instruction-set simulation.

At the output of the backend, behavioral hardware and software processor models in the TLM are synthesized down to their component- and task-level implementations ready for further synthesis and manufacturing. On the hardware side, both application algorithms and bus interfaces are refined into synthesizable VHDL or Verilog RTL models. On the software side, code for application tasks, middleware and bus drivers is automatically synthesized into final target binaries ready for download into the processors.

In addition to VHDL or Verilog descriptions and binary images for each hardware or software processor, respectively, an implementation model of the system is generated that allows for co-simulation of hardware RTL models with software instruction-set simulators (ISSs) running final target binaries. As a result, the pin- and cycle-accurate implementation model realizes a netlist MoS and a MoP that is based on a CAPM.

[...]

SCE follows a Specify-Explore-Refine methodology [188]. The design process starts from a model specifying the desired functionality (Specify). In each following design step, the designer first makes necessary design decisions by exploring the design space (Explore). SCE then automatically generates a new model at the next lower level of abstraction by integrating decisions and database component models into the design (Refine). As such, through a gradual, stepwise refinement process, SCE automatically generates models successively at lower levels of abstraction and with an increasing amount of implementation detail.

[...]

[The] SCE system design front-end internally consists of four design steps: *architecture and scheduling exploration* for design of system computation, followed by *network exploration* and *communication synthesis* for design of system communication.

[...]

- During *architecture exploration*, the processing platform (PEs and memories) is defined and the computational aspects of the specification (behaviors and variables) are mapped onto that platform. During scheduling exploration, the order of execution on the inherently

sequential PEs is determined. Behaviors can be statically scheduled and grouped into sequential tasks, and remaining concurrent tasks are dynamically scheduled on top of a real-time operating system (RTOS).

- During *network exploration*, the system communication topology (busses, CEs and their connectivity) is defined, and the given end-to-end communication channels are mapped and routed over that network.
- During *communication synthesis*, point-to-point links in each network segment are implemented over the actual bus medium, and pin- and bit-accurate parameters, such as bus addresses and interrupts, are selected.

Finally, in the back-end, hardware and software synthesis of each synthesizable or programmable PE and CE is performed. Hardware synthesis follows an interactive and automated high-level synthesis process to take behavioral hardware models down to structural RTL descriptions. For software synthesis, SpecC code for application software, middleware, drivers and interrupt handlers is generated, cross-compiled, and targeted towards and linked against real-time operating system (RTOS) to create final target binaries. [...]" ([28] © 2009 IEEE)

4.6.3 SystemCoDesigner

"The goal of SystemCoDesigner is to automatically map applications written in SystemC to a heterogeneous MPSoC platform. By automating as many design steps as possible, an early evaluation of different design options is [possible] [31]. [...] In a first step, the designer writes an actor-oriented application model using SystemC. In a second step, different hardware accelerators are automatically generated for actors and stored in a component library. This library also contains other synthesizable IP cores like processors, buses or memories. The designer defines an MPSoC platform model from resources in the component library as well as mapping constraints for the actors, resulting in a system-level specification. An automatic design space exploration trades off several [...] design objectives. From the set of optimized solutions, the designer selects promising implementations for rapid prototyping. For this purpose, design decision leading to the optimized solution are represented as structural TLM. For rapid prototyping, hardware accelerators are synthesized to the RT level and software is compiled to match the ISA of selected processors.

[...]

Currently, SystemCoDesigner supports the design of streaming applications only. These applications are typically modeled by help of dataflow graphs where [nodes] represent actors and edges represent data dependencies. Due to the complexity of many streaming applications, they often cannot be modeled as static dataflow graphs [195], [184], where consumption and production rates are known at compile time. Rather they are described as a combination of static and dynamic dataflow models, e.g., Kahn Process Networks [200].

[...] SystemCoDesigner assumes that the application model is written in SystemC and represents a dataflow model, i.e., SystemC modules (actors) only communicate via SystemC FIFO channels and their functionality is implemented in a single SystemC thread. Such input descriptions can be transformed into a special subset of SystemC called SysteMoC [31]. An application modeled in SysteMoC resembles the FunState MoC (Functions driven by State machines) [156] that allows to express non-deterministic dynamic dataflow (DDF) models.

A SysteMoC model is composed of SysteMoC actors that communicate via queues with FIFO semantics. Each SysteMoC actor is defined by a finite state machine (FSM) specifying the communication behavior and methods controlled by the finite state machine. If activated by the FSM, these methods are executed atomically and data consumption and production is only performed after computing a method. [...] Furthermore, constant methods, called *guards* [...], can be used to test values of internal variables and data in the input channels. If predicates annotated to a state transition evaluate to true, this transition can be taken and annotated *action* methods [...] will be processed atomically.

SysteMoC actors can be transformed into both hardware accelerators and software modules [31]. The latter one is achieved by straight forward code transformations, whereas the hardware accelerators are built by help of Forte Cynthesizer [211]. This allows for quick extraction of important performance parameters like the achieved throughput and the required area [,]

which are used to calibrate the system-level specification. The generated hardware accelerators (synthesizable RTL code) are stored in the component library. This component library contains further synthesizable IP cores including processors, buses, memories, etc. The MoA is a heterogeneous MPSoC platform [,] which is specified by instantiating and connecting cores from the component library. Furthermore, the designer has to specify mapping constraints for each SysteMoC actor. Later, design space exploration is performed to find sets of optimized solutions.

From the set of optimized solutions [,] the designer selects any MPSoC implementation best suited for his needs. Once this selection has been made, the last step of the proposed ESL design flow is the rapid prototyping of the corresponding FPGA based implementation in terms of model refinement. For this purpose, the resulting platform is assembled. Moreover, the program code for each processor is generated according to the binding of the actors. This results in a TLM, which is the MoS used as implementation representation by SystemCoDesigner. In order to generate [...] software schedules, SystemCoDesigner supports the automatic classification of actors into synchronous or cyclo-static dataflow [56] and clustering static actors bound to the same processor into a single dynamic actor [38]. Finally, the implementation is compiled into an FPGA bit stream using the Xilinx Embedded Development Kit (EDK) [95]. [...]

[SystemCoDesigner is capable to explore the design space automatically.] For this purpose, it optimizes the implementation of the streaming application while considering several objectives simultaneously, e.g., latency, throughput, area and power consumption. While area consumption is assumed to be a linear cost function, timing and power estimation requires a simulation-based performance evaluation during exploration.

SystemCoDesigner generates task-accurate MoPs (TAPM) automatically from the SysteMoC model and the performance values annotated in the input model [31]. For this purpose, the MPSoC platform model is translated into a so called *virtual architecture* using again SystemC. The performance evaluation is done by linking the SysteMoC model to the virtual architecture. Each invocation of an action of an actor is then relayed to the virtual component the actor is bound to. The virtual component then blocks the actor's execution until the estimated execution time of the action and possible other preemption times are expired.

Beside evaluating a single design point, design space exploration is responsible for covering the search space. In order to perform decision making automatically, SystemCoDesigner translates the input model into a Pseudo Boolean (PB) formula. The variables of this formula encode the resource allocation, the actor binding, the queue mapping, and the routing of transactions on the communication structure. Each variable assignment satisfying this formula corresponds to a feasible implementation of the application. A Pseudo Boolean solver is used to identify these solutions [31]. The optimization is performed using a Multi-Objective Evolutionary Algorithm. [...]" ([28] © 2009 IEEE)

4.6.4 Metropolis

"Metropolis [130] is a modeling and simulation environment based on the platform-based design paradigm [71]. [...]

Metropolis provides a general, proprietary meta-model language that is used to capture separate models for "functionality" (behavioral model), "architecture" (platform model) and their "mapping" (binding and scheduling). The meta-model employs a fundamental event-based execution model with concepts of concurrent processes communicating through channels (called media), including associated constraints and quantities. In a similar manner to other system-level languages, functionality is described in the form of event-driven process networks that are general in the sense that many classes of MoCs can be represented. In addition, functionality can be annotated with extra-functional constraints. The architecture is defined following [a] MoA that uses processes and media to describe available resources (e.g. tasks) and services (e.g. CPUs, memories or buses), respectively. Quantities can be associated with the architecture to define [a] MoP at the level of tasks (TAPM). Finally, given a specification in the form of functionality and architecture, synthesis or refinement is performed by defining [a] MoS as a mapping between the two through a set of additional constraints synchronizing their event execution.

Metropolis itself does not define any specific design tools but rather a general framework and language for modeling with support for simulation, validation and analysis of models. Metropolis includes a front-end for parsing of meta-models and a back-end for translation of meta-models

into C++/SystemC simulation code. In addition, several back-end point tools have emerged for scheduling, communication design, verification, and hardware synthesis [212].” ([28] © 2009 IEEE)

4.6.5 Koski

“The Koski design flow [80] provides a single infrastructure for modeling of applications, automatic architectural design space exploration, and automatic ESL synthesis, programming, and prototyping of selected MPSoCs. Koski’s design flow starts with the capturing of requirements for an application and architecture, including design constraints, such as the overall maximum cost. Subsequently, the functionality of the system is described with an application model in a UML design environment (using the Statecharts MoC to describe the actual functionality) and verified with functional simulations. The architecture model consists of components [,] which are taken from a platform library, targeting the construction of heterogeneous, bus-based MPSoCs (MoA). The relationship between application and architecture models is described with a mapping model.

The UML interface handles the transformation of application and architecture models to an abstracted model for fast architecture exploration. Particularly, the application model is transformed to an abstract process network model. In addition, the UML interface can back-annotate the UML design with performance information obtained from lower-level simulations. Finding a good application-to-architecture mapping is carried out during a two-phase automatic architecture exploration step consisting of static and dynamic (i.e., simulative) exploration methods using a TAPM MoP. For controlling the architecture exploration, the designer constrains the design space by defining the platform parts that can be used as well as the allowed mapping combinations. In addition, the designer specifies the constraints for performance, area, and power.

In the last step, the parts of the UML description that were mapped to processors during the architecture exploration are passed to the automatic code generation. The generated low-level software code and the RTL descriptions (i.e. a netlist MoS) of the component instances from the platform (derived from Koski’s platform library) are then combined for physical implementation. This stage also handles the real-time operating system (RTOS) integration, software executable generation, and hardware synthesis.” ([28] © 2009 IEEE)

4.6.6 PeaCE/HOPES

“PeaCE (Ptolemy extension as a Co-design Environment) [46] is an ESL synthesis framework for multimedia applications. Starting from a Ptolemy II application model, it provides a seamless co-design flow from functional simulation to system synthesis and prototyping. Although Ptolemy supports the hierarchical combination of many different Model of Computation, PeaCE restricts the input model to extension of synchronous dataflow and extended finite state machines. In PeaCE, the application is modeled by a task graph where tasks are either signal processing tasks or control tasks. Signal processing tasks are modeled through synchronous piggybacked dataflow, a dataflow model with control token. Control tasks are modeled by flexible finite state machines (hierarchical state machines without state transitions crossing hierarchy boundaries).

For functional simulation of the application model, PeaCE provides an automatic C code generation. For system synthesis, the architecture platform is specified by a list of processors and synthesizable IP cores resulting in a heterogeneous MPSoC Model of Architecture. The design space exploration is a two-phased: In a first step the resource allocation and task binding is performed. During this step, communication overhead is assumed to be proportional to the amount of consumed and produced data. The objective of this step is to minimize system cost under timing constraints. In the second step, the communication architecture exploration, that is bus and memory allocation is performed. For this purpose, communication and memory traces are generated for those solutions fulfilling the timing constraints in the first step. Design space exploration in PeaCE can be performed automatically or manually and is guided by an instruction set accurate performance model. After design space exploration, optimized MPSoC implementations can be prototyped either using a co-simulation environment or FPGAs. In both cases, the Model of Structure is a Netlist representing the design decisions.

Recently, a new framework called HOPES has been proposed as enhancement to PeaCE [50]. The main focus is on generating MPSoC software and overcome the limitations of OpenMP and MPI. Its input model is called CIC (Common Intermediate Code). A CIC models consists of two parts: The task code defines each task by the three methods `init()`, `go()`, and `wrapup()`. Inter-task communication or communication to the environment is established by help of several APIs. The second part is the architecture information, including the platform definition and additional constraints. The task code of a CIC model can be either written manually or automatically generated from PeaCE models.

A CIC translator transforms a CIC model into optimized software for the processors in the MPSoC platform. For this purpose, the API calls must be replaced by platform specific code, interface code for hardware accelerators has to be generated, and scheduling of tasks bound to the same processor has to be performed. Optionally, an OpenMP compiler can be used for optimization." ([28] © 2009 IEEE)

4.6.7 Summary & Discussion

	Specification		Implementation			Decision Making		Refinement	
Approach	MoC	MoA	MoS	MoP	DSE	Comp	Comm	Comp	Comm
Daedalus	C/ KPN	HeMPSoC	Netlist	TAPM/ ISAPM	●	●	○	●	○
Koski	Statecharts	HeMPSoC	Netlist	TAPM	●	●	○	●	○
Metropolis	PN	HeMPSoC	TLM	TAPM		○		○	
PeaCE/ HoPES	DDF/FSM	HeMPSoC	Netlist	ISAPM	○	○		●	○
SCE	C/ PSM	HeMPSoC	TLM/ Netlist	TAPM/ CAPM				●	●
SystemCo-Designer	C++/ DDF	HeMPSoC	TLM	TAPM	●	●	●	●	
OSSS	C++/ PSM	HeMPSoC	TLM/ Netlist	TAPM/ CAPM	○			●	●

DDF	Dynamic Dataflow	TLM	Transaction-Level Model
(K)PN	(Kahn) Process Network	TAPM	Task Accurate Performance Model
PSM	Program State Machine	ISAPM	Instruction Set Accurate Performance Model
HeMPSoC	Heterogeneous, Bus-Based Multi-Processor System-On-Chip	CAPM	Cycle-Accurate Performance Model

Table 4.2: Classification of different ESL synthesis approaches (based on [28])

Table 4.2 compares the OSSS methodology, using the classification scheme of [28]. The different ESL synthesis approaches differ in their supported input specifications and resulting output models. OSSS enables the usage of C++ and Program State Machines (PSM). For synthesis, the PSM parallel composite behaviors (PAR and PIPE) need to be refined to sequential actors and Shared Objects for coordination and synchronization. The output of the synthesis is a Task Accurate Performance model with annotated estimated execution times for the software part and a Cycle-Accurate Performance Model of the hardware part. Compared to other methodologies, OSSS has no support for automatic decision taking. Design Space Exploration (DSE) is not explicitly included in the OSSS framework, but supported by the methodology through its plug-and-play concept:

- Shared Objects are prepared for handling and arbitrary number of client processes. These can be added and removed during Application Layer exploration without the need to modify the Shared Object implementation.
- Shared Object scheduling algorithms can be easily exchanged with minimal configuration parameter adaptation (i.e. client access priority annotations).
- Shared Object communication link (binding between client port and Shared Object at Application Layer) to RMI Channel re-mapping enables to explore different communication

topologies: ranging from single shared bus to dedicated point-to-point connections only.

- RMI Channels are realized as hierarchical channels and thus their physical implementation can be exchanged independently from the RMI protocol layers. For shared buses the scheduler can be exchanged with minimal configuration parameter adaptation (cp. Shared Object scheduler exchange), and for point-to-point channels the physical channel width can be configured.

Computation and Communication refinement is explicitly supported by the OSSS methodology, even though it is currently not implemented in a graphical tool. Like most of the presented ESL frameworks OSSS also targets FPGA platforms for synthesis.

4.7 Contribution of this work

The main contribution of this thesis is to provide an efficient design methodology for mapping object-oriented applications to embedded System on Chip architectures, including automatic communication interface refinement and synthesis. The main distinguishing feature of this thesis is the seamless support of application-level method call communication on all levels of abstraction, down to an FPGA platform. Already presented in Section 1.3, the main contributions are now further broken down:

1. Definition of a **multi-layer executable parallel object-oriented application description** that supports **custom application-level method call communication**.

   Definition and integration of an executable object-oriented Program State Machine (OOPSM) model within the OSSS Behavioral Layer [43]. So far, PSM modeling has only been supported in SpecC. The support of PSMs enables OSSS to capture a wide range of Models of Computation (MoC) relevant for the design of embedded systems (see Section 3.6).

   Executable actor-oriented OSSS Application Layer consisting of Actors, Objects and Shared Objects. Custom application-level method calls are supported by custom Shared Object interface definition and implementation, based on the basic communication channel principle of Interface Method Calls (IMC) [65, 22].

   Shared Objects in this work have the following properties, which are different from previous work [135, 97] as described in Section 4.2:

Table 4.3: Shared Object modeling contributions

<table>
<tr><th>previous work [135, 97]</th><th>this work</th></tr>
<tr><td colspan="2">A Shared Object consists of a set of data members (local variables) together with a set of member functions or methods which operate on those variables.</td></tr>
<tr><td colspan="2">The data members of a Shared Object must be used only by its methods.</td></tr>
<tr><td>The set of methods provided by a Shared Object for external use is called its interface. Interaction with a Shared Object is only allowed through this interface.</td><td>The set of methods, explicitly declared in an interface class, are the interface of a Shared Object. A Shared Object implements these interfaces. Interaction with a Shared Object is only allowed through its interfaces.</td></tr>
<tr><td colspan="2">Concurrent accesses to a Shared Object are synchronized by some mechanism. Any such synchronization mechanism ensures that concurrent requests leave a shared object always in a consistent state.</td></tr>
<tr><td>Enforcing mutual exclusive execution of all concurrently requested methods is one, but not the only possible solution that satisfies this requirement.</td><td>Mutual exclusiveness is enforced.</td></tr>
</table>

continued on next page

Table 4.3: Shared Object modeling contributions – *continued*

<table>
<tr><th>previous work [135, 97]</th><th>this work</th></tr>
<tr><td colspan="2">A Shared Object is passive, which means it is not able to initiate any action by itself without being externally triggered.</td></tr>
<tr><td>Methods of a Shared Object can have an associated guard condition. The guard condition may only involve members of a Shared Object *and* input parameters of the method.</td><td>Interface methods of a Shared Object can have an associated guard condition. The guard condition may *only involve members of a Shared Object.* The guard condition is *not* dependent on the input parameters of the method.</td></tr>
<tr><td colspan="2">Calls to guarded methods may be accepted, only if the guard evaluates to true. Otherwise the call will be delayed up to that point. The guard mechanism ensures, that the guard condition does not change between its evaluation and the execution of the method it is associated with.</td></tr>
<tr><td>A Shared Object is declared in a scope, where it could be shared between concurrent processes.</td><td>A Shared Object can be placed anywhere in the design hierarchy. Client processes access the Shared Object through dedicated ports by explicit port to Shared Object binding. This binding can be performed through the design hierarchy by port to port bindings. A port is strongly typed by an interface of the Shared Object. This strongly typed port can only access the methods declared by this interface. Note that a Shared Object can implement multiple interfaces.</td></tr>
<tr><td colspan="2">No request to a method gets lost. Requests by the same client must be served and completed in the same order they were issued. Any parameters returned by a method invoked on a shared object must be available before their first use.</td></tr>
<tr><td colspan="2">No method of a shared object must invoke a method of another shared object but itself. Nested method invocations are executed unsynchronized (and unguarded).</td></tr>
<tr><td colspan="2">A guarded method execution, once started, can not be interrupted.</td></tr>
<tr><td>The exact latency of a guarded method execution and the total delay caused by a request must be regarded as unknown.</td><td>On Application Layer, the exact latency of a guarded method execution and the total delay caused by a request is unknown a priori, but becomes visible during simulation and can be statically analyzed (assuming deterministic client access patterns).</td></tr>
</table>

Furthermore, mapping and refinement rules from OSSS Behavioral Layer to OSSS Application Layer models will be defined.

2. Definition of an **object-oriented model of target platform** that represents processing, communication and memory resources.

   In previous work [162, 135, 97] only modeling of custom hardware has been considered. This work aims at modeling custom software, hardware and their communication. This work supports the definition of a target platform consisting of pre-existing processing, memory and communication resources and their interconnection. This structural representation is required, to separate the application description on the OSSS Behavioral and Application Layer from the platform resources. Focusing on communication, the main contributions of this work are:

OSSS Channels [77, 78] enabling the a self-configurable, executable description of on-chip bus and point-to-point communication networks. Thus extending the work of [153] to multi-master buses and flexible arbitration.

OSSS RMI Channels [65, 22] are hierarchical channels, encapsulating OSSS Channels by providing a Remote Method Invocation protocol for HW/SW and HW/HW communication.

OSSS Shared Object Sockets [65] for the integration and connection of Application Layer Shared Objects with OSSS RMI Channels to express communication refinement of the Actor port to Shared Object binding.

3. Definition of a relation to express computation, communication and memory **mappings of the application to a "bare metal[6]" target platform, retaining application-level method calls** through Remote Method Invocation (RMI) techniques.

   With the ability to refine an OSSS Behavioral Layer model to an OSSS Application Layer model and the description of a platform computation, memory and communication resource model, this missing link is a flexible mapping of Application Layer modeling elements to the platform model.

   The platform model in OSSS is called Virtual Target Architecture. This work defines mapping and refinement rules to transform an executable Application Layer model into an executable Virtual Target Architecture Layer model. This combined model is a cycle accurate representation of the platform's communication resources. The timing granularity of the custom software and hardware functionality can be refined to cycle approximate representation (for software) and cycle accurate representation for hardware. However, this computation timing refinement is not in the scope of this work. This is also one of the main reasons why this work only covers "bare metal" target platforms. I.e. no operating systems are considered in this work.

4. Definition of operational semantics and implementation of a pre- and post-target platform mapping **simulation for checking functionality and timing**.

   As mentioned before, all layers of OSSS can be simulated and enable validation of the implemented functionality and timing (depending on the actual computation timing refinement). For this purpose, all OSSS modeling elements have been implemented on top of SystemC [44, 42]. Our contribution distinguishes between:

   pre platform mapping simulation relates to the simulation of the OSSS Behavioral Layer [43] and the OSSS Application Layer models.

   post platform mapping simulation relates to the simulation of the OSSS Virtual Target Architecture Layer model.

   The OSSS Behavioral Layer simulation library can be obtained from `http://system-synthesis.org/_media/osss-behaviour-0.0.2.tar.gz` and the OSSS Application and Virtual Target Architecture Layer simulation library can be obtained from `http://system-synthesis.org/_media/osss-2.2.1.tar.gz`[7].

5. Definition of timed automata formal model of pure application and target platform mapped application model to enable **analysis of safety** (deadlock, method call duration, mutual exclusiveness of method calls, buffer size limitations) **and liveliness properties** (guarantee that a method call will be served).

   Restricting Shared Objects in the usage of guards (i.e. guards are independent on the arguments of interface methods[8]) an analytical model of the OSSS Application Layer and Target Architecture, based on Timed Automata can be derived. In previous work [21, 10] an OSSS to real-time task network to Timed Automata mapping has been presented.

[6] *Bare metal* (or *bare machine*), in computer terminology, means a computer without an operating system.
[7] This is a public release, which contains OSSS extensions not covered by this work.
[8] otherwise a static analysis might become infeasible

A combination with the basic concepts presented in [4], led to a direct representation of OSSS models in UPPAAL Timed Automata.

An OSSS to UPPAAL timed automata representation is provided including the analysis of the following[9]:

safety properties of the kind

- System never runs in a deadlock.
- Completion of a method call on a Shared Object never takes longer than a certain time limit.
- Execution of methods of the same Shared Object are always executed mutually exclusive.
- The total amount elements in an array of the maximum value of a variable inside a Shared Object is bounded.

liveness properties of the kind

- When a method call on a Shared Object is requested it will eventually be severed.
- When a method call on a Shared Object is completed another method call eventually completes.

6. Definition of synthesis semantics and **proof-of-concept synthesis of mapped application on FPGA target platform** (enabling memory space and hardware area analysis) supporting RMI synthesis for hardware/hardware and hardware/software communication.

   For synthesis of the OSSS Application Layer + Virtual Target Architecture Layer model to an FPGA, a combination of Xilinx EDK (for the hardware platform and software synthesis), FOSSY (for the SystemC and Shared Object to VHDL synthesis) and Xilinx ISE (for the VHDL to FPGA configuration bit stream synthesis) will be proposed. The specific contributions of this work are:

   - OSSS Shared Object to VHDL synthesis with a flexible communication channel interconnect [22], compared to [135, 97], where the client to Shared Object interface was fixed.
   - OSSS Virtual Target Architecture Layer to Xilinx FPGA platform synthesis. This includes structural analysis of the OSSS Virtual Target Architecture Layer model including the OSSS Application Layer mapping relationships, intermediate model representation and Xilinx EDK project output generation.
   - OSSS RMI Channel synthesis including custom hardware client interface synthesis, configurable software driver library and basic run-time support, and Shared Object RMI controller synthesis.
   - Bit width configurable OSSS point-to-point channel synthesis.

[9] The properties have been implemented, but can be extended

CHAPTER 5

Methodology, Modeling Elements and Operational Semantics

5.1 Introduction

This chapter presents the proposed object-oriented methodology, defines its modeling elements and their operational semantics. Figure 5.1 (already introduced in the summary of Section 3.5) describes the design methodology used in this work. It is a combination of a platform-based design methodology, called system-level methodology, and an FPGA-based design methodology. For more information about the Y-Chart and the associated design methodologies further background information can be found in Section 3.5.

In Section 3.4 Models of Computation (MoC), Models of Architecture (MoA), Models of Structure (MoS) and Models of Performance (MoP) have been introduced and connected with the X-Chart to describe model refinement and synthesis as depicted by the concentric arrows from the behavior to the structure and physical views.

As depicted in Figure 5.2 certain combinations of MoC, MoA, MoS and MoP have been clustered in four different layers: *Behavioral* (A), *Application* (B), *Virtual Target Architecture* (C)(D)(E) and *Implementation Layer* (F). Red arrows indicate refinement/synthesis steps described in this thesis. Dotted arrows indicate refinement/synthesis steps supported by the methodology but not described in this work.

In this chapter the models (A), (B), (D) and (F), and the associated design flow are described. The following section (5.2) gives a general overview of the abstraction layers supported by the methodology, from the pure functional design entry, down to the target platform implementation.

In Section 5.3 the general *Object Model* used in this thesis is described. In Section 5.4, Section 5.5 and Section 6.5 the Behavioral Layer (A), Application Layer (B) and Virtual Target Architecture Layer (D) modeling elements and their operational semantics is described.

After each of the detailed modeling element descriptions for the Application and the Virtual Target Architecture Layer in Section 5.5.2 and Section 5.6.2, mapping rules to establish the design flow for the transitions from one abstraction layer down to another are presented in Section 5.5.5 and Section 5.6.3. These sections also discuss the necessary design decisions to drive the mapping process. For the analysis of functional and extra-functional properties simulative and analytic techniques can be used on the proposed abstraction layers. Simulative techniques will be described in more detail in Chapter 6, while static timing analysis techniques using timed automata is described in the operational semantics sections 5.4.3 and 5.6.4.

In Section 5.7 only a brief overview of the implementation on the Target Platform Layer (F) is described. More details about the Virtual Target Architecture to Target Platform Layer mapping and synthesis step are described in Chapter 7.

This chapter closes with a summary and discussion of the requirements on the methodology and flow as demanded in Chapter 2.

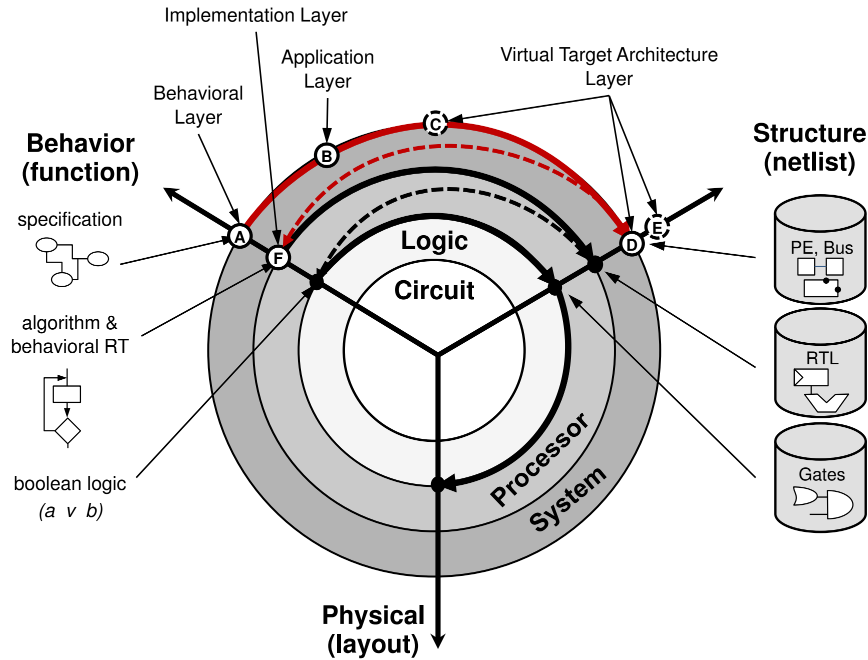


Figure 5.1: Proposed design methodology: Combination of System-level and FPGA-based design methodology as used in this work. The design points (A) - (F) correspond to the X-Chart points in Figure 5.2

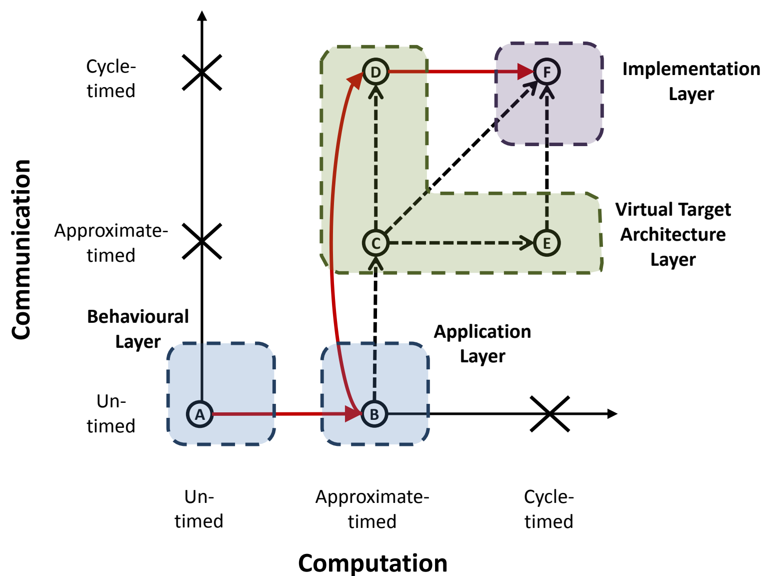


Figure 5.2: Overview of modeling layers for the proposed methodology. Red arrows indicate refinement/synthesis steps described in this work. Dotted arrows indicate refinement/synthesis steps not described in this work.

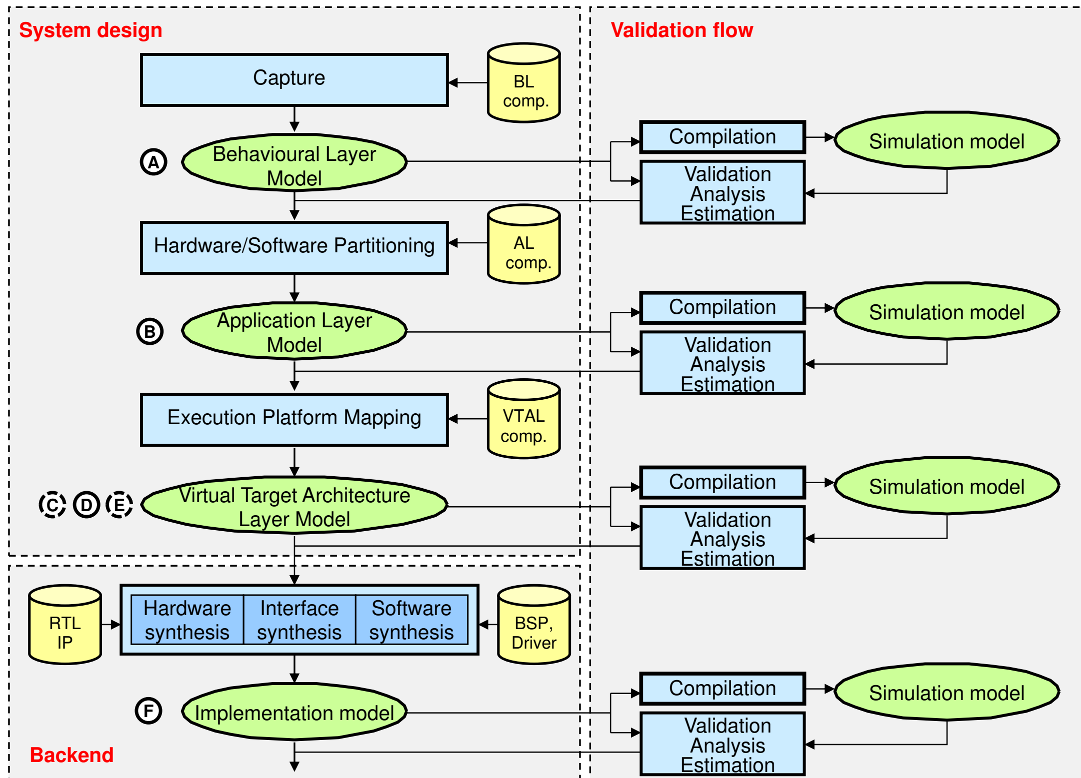


Figure 5.3: Design flow (based on scheme in Figure 3.11) used in this work (adapted from [150])

5.2 Abstraction Layers

This sections gives an overview of the proposed abstraction layers and their integration into a design flow as introduced in Section 3.5.1. Figure 5.3 shows the design flow (based on the scheme as introduced in Figure 3.11) with the abstraction layers used in this work. In this chapter we focus on the description of the *System Design* and the general usage of the *Validation Flow* and the *Back-end*. The simulation models of the Behavior (A), Application (B) and Virtual Target Architecture (D) Layer will be described in Chapter 6. The back-end including hardware, software and interface synthesis to end up with an implementation model will be described in Chapter 7.

The system design, as shown in Figure 5.3 describes the refinement path from the initial functional design model (A) to a target platform mapped functional model (D) that can be synthesized into an implementation model (F) for the supported target platforms (cp. red lined in Figure 5.1 and Figure 5.2).

In a first step the functionality of the embedded system is captured as an executable model. The capturing of functionality is performed in a structured way by using the provided modeling elements of the Behavioral Layer. The Behavioral Layer modeling elements are hierarchical behaviors and a pre-defined set of communication channels that can be composed following the Program State Machine semantics as introduced in Section 3.6.2. The initial Behavioral Layer model is untimed and used to expose the maximum degree of parallelism the functional system description offers (or can be achieved with a reasonable amount of modeling effort). Execution and profiling of this model can be applied to functional and extra-functional metrics to identify relevant and computationally intensive elements. This information is used for the first refinement step from the Behavioral (A) to the Application (B) Layer.

The step from the Behavioral to the Application Layer consists of following major tasks:

1. grouping and re-scheduling of behaviors to be mapped onto processing elements of the target platform

2. grouping and re-scheduling of communication channels to be mapped onto *Shared Objects*
3. back-annotate estimated execution times at leaf behaviors, reflecting the execution time/-duration of this behavior running on the targeted processing element
4. annotate estimated execution times to Services of Shared Objects, reflecting the time/duration of the communication service being executed on the targeted processing, memory an/or communication element

The purpose of the re-grouping, re-scheduling and timing annotation of behaviors and communication channel is to obtain an initial performance model of a certain platform mapping or hardware/software partitioning.

The main modeling element for computation on the Application Layer is called an *Actor* An Actor consist of a single or multiple processes (depending on its kind, either representing a (sequential) *Software Task* or a parallel *Custom Hardware Module*). Communication and synchronization between Actors is modeled by *Shared Objects*. These are special communication objects that provide a method interfaces for communication and guarantee a consistent access of an arbitrary number of concurrent Actors.

In summary, on the Application Layer we specify the function, logical structure, and an approximate time response of the system. Profiling results from analysis of the Behavioral Layer model can be annotated to the Application Layer model to obtain an approximate-timed behavior. Also back-annotation approaches, where the execution of Leaf Behaviors are profiled on their expected target processing elements, are possible, but not further discussed here. Besides the functional correctness of the system, the Application Layer offers an easy evaluation of design alternatives (e.g. HW/SW partitioning, scheduling, communication structures, and data locality). Profiling of different Application Layer model alternatives with regard to their performance can be accomplished easily since the component's allocations and scheduling can be changed quickly. Analysis of the executable Application Layer model in early design phases can help to detect and resolve bottlenecks in the logical structure. This might result in the relocation of timely critical computations from SW to HW or in reorganizing complex computations in pipeline structures to enhance the throughput.

The Application Layer Model is executable and abstracts from platform details regarding the communication between Actors and Shared Objects. To include this information into the system mode, the Application Layer gets further refined and mapped to a component model of the targeted implementation or execution platform.

The *Virtual Target Architecture Model* (VTA Model) adds implementation details of the target architecture with a special focus on the target platform inter processing element communication network. Communication links between Actors and Shared Objects of the Application Layer model are mapped onto cycle accurate communication channels. A flexible model for cycle accurate bus and point-to-point connections that enables the description of different protocols and data bandwidths enables exploring the impact of different protocols, data widths and arbitration schemes on the timing behavior of Application Layer model.

The resulting VTA model's communication is cycle accurate including the approximate timing annotation inside Actors. Moreover, the Application Layer mapped on the VTA Layer model is again an executable model that contains all relevant information required to start the back-end synthesis flow towards an implementation model.

The VTA model is the input for the *Fossy* (**F**unctional **O**ldenburg **S**ystem **Sy**nthesiser) synthesis tool. It generates the overall system architecture, synthesizable VHDL for each hardware component, and C/C++ code for each software task. For the software parts a driver API and for the hardware parts a bus interface is automatically generated. Depending on the chosen platform, different so-called architecture description files can be generated. Special properties of different target platforms require adoptions of the synthesis process, e.g. for embedding special IP blocks or the generation of 3rd party tool specific configuration files.

5.3 Object Model

In this work we will focus on the general-purpose object-oriented C++ programming model [14, 8]. A comparison between different object oriented programming models can be found in [176].

Due to the requirements **M1** (Single modeling language to describe HW and SW) and **M4** (Synthesizable HW/SW partitioned model) from Section 2.5, the generic C++ programming model is restricted to a reduced object model that will be introduced in the following subsections.

5.3.1 Basic Types

Definition 5.3.1.1 (Basic Types):
For simplicity we assume only the three basic types:

1. ***integer*** *denotes the set of numbers that can be written without a fractional or decimal component. We further distinguish between*
   - ***unsigned integer*** *denotes the set of positive natural numbers from the interval* $[0, 2^n - 1]$*, where* n *is the number of bits in two's complement machine representation*
   - ***signed integer*** *denotes the set of positive and negative natural numbers from the interval* $[-2^{n-1}, 2^{n-1} - 1]$*, where* n *is the number of bits in two's complement machine representation*
2. ***Boolean*** *denotes the set* $\{\textbf{\textit{false}}, \textbf{\textit{true}}\}$
3. ***bit*** *denotes the set* $\{0, 1\}$

□

5.3.2 Other types

In the reference implementation of this methodology based on SystemC more types are supported. Especially for explicit description of hardware, multivalued logic types can be used, but for our purposes it will not be of any use to include theses types in this basic definitions. A full list of supported and synthesizable types can be found in Appendix F.

5.3.3 Array

Definition 5.3.3.1 (Array):
We define an ***Array*** *as* $T[n]$*, where* T *denotes some basic type, class or another array of the same type and* $n \geq 1$ *is the size of the array.*

The index function $T[n] \times \textbf{\textit{integer}}_{\geq 0} \rightarrow T$ *allows to access the* n *elements of the array starting with index of 0.* □

In the following we only consider single dimensional arrays. From an array of basic or class type T with $k \geq 2$ dimensions and sizes $s_k \geq 1$ for all dimensions, a single dimensional array of type T with size $\prod_{j=1}^{k} s_j$ can be constructed.

An index remapping function from k to one dimension is given as

$$\textbf{integer}_{[0,s_1]} \times \cdots \times \textbf{integer}_{[0,s_k]} \rightarrow \textbf{integer}_{[0,\prod_{j=1}^{k} s_j]} : ((\ldots((i_1 \cdot s_2 + i_2) \cdot s_3 + i_3) \cdot \ldots) \cdot s_k + i_k)$$

with i_j, $j \in [1, k]$ indexes for each dimension and sizes $s_k \geq 1$ for each dimension.

5.3.4 Class

Definition 5.3.4.1 (Set of Classes):
The set of all classes $\mathcal{C}$ *is* $\wp(C \times IF)$*, where* C *is a class and* IF *is an interface class.* □

Definition 5.3.4.2 (Class):
A ***Class*** *is a tuple* $C = [c_{parent}, State, Method]$*, with*

1. *a single base class* $c_{parent} \in \mathcal{C}$ *(single inheritance),*
2. *a state vector* $State = T_0 \times \cdots \times T_n$*, with* $n \geq 0$ *where* $T_0, \ldots T_n$ *denote some basic types, classes, or arrays,*

3. *and a set of member functions* $Method = \{m_0, \dots m_m\}$ *with* m_i, $i \geq 0$ *of the following kinds:*

$$\begin{aligned} name\colon \textbf{void} \to \textbf{void} &\quad \text{no arguments and no return type} \\ name\colon \textbf{void} \to T &\quad \text{no arguments and return type } T \\ name\colon T_0 \times \cdots \times T_n \to \textbf{void} &\quad \text{arguments } T_i,\ i \geq 0 \text{ and no return type} \\ name\colon T_0 \times \cdots \times T_n \to T &\quad \text{arguments } T_i,\ i \geq 0 \text{ and return type } T \end{aligned}$$

□

Each class is required to have at least a default constructor which is a member function with the same name as its class and without any arguments. The constructor is required to initialize the state of the class.

If a class shall be copyable and assignable the following two member functions also need to be defined:

Copy constructor Construct all the object's members from the equivalent members in the copy constructor's parameter

Assignment operator Assign all the object's members from the equivalent members in the assignment operator's parameter

For simplicity we assume for our classes the programming idiom called "Resource Acquisition Is Initialization" (RAII). In RAII resource acquisition is performed by constructing an appropriate object. The object manages the lifetime of the resource and the object's destructor ensures the resource gets "un-acquired". The destructor is another special member function that gets called when the lifetime of the object ends. Only dynamically allocated memory (if its usage is allowed) needs to be released here again. If no dynamically assigned memory is used the destructor can be omitted.

Normalization/Flattening For the representation of classes in a linear memory array or in custom hardware [162] the state vector needs to be represented accordingly as well as member functions and Methods accessing this state vector. We call this linearization *Normalization* or *Flattening*. Classes are normalized or flattened by a state vector concatenation of all parent class types. Consider a class inheritance relationship $c_n \to c_{n-1} \to \cdots \to c_0$, with $n \geq 1$ and c_0 being the root class with no more parent class(es). Then the flattened state Type of class c_n is $c_0.State \times \cdots \times c_{n-1}.State \times c_n.State$. The flattened set of Methods of class c_n can be described as $c_n.Method \uplus \cdots \uplus c_{n-1}.Method \uplus c_0.Method$, where $\uplus$ is a modified $\cup$ with $A, B \in C$:

$$A \uplus B = \begin{cases} A \cup B & \text{if } A \cap B = \emptyset \\ A \cup (B \setminus A \cap B) \cup classname_prefix(classof(B), A \cap B) & \text{otherwise} \end{cases}$$

The function $classname_prefix\colon C \times C.Method \to C.Method$ adds the name of the class C as a prefix to all Method names in the set $C.Method$. In other words, the flattened set of Methods is the union of all Methods. For Methods that have the same name and type of signature only the method of the class which is the closest to c_n is taken into the flattened set. The remaining methods get a prefix of their class name and are added to the flattened set.

For these normalized or flattened classes all state accesses and invocations of base-class methods are assumed to be adapted to accesses to the flattened state vector as well as flattened set of methods.

For more details on synthesis of digital circuits from object-oriented specifications see [162].

5.3.5 Interface class

Definition 5.3.5.1 (Interface Class):
An ***Interface Class*** *is a tuple* $IF = [\overline{c}_{parent}, Method]$, *with*

1. *a list of base classes* $\overline{c}_{parent} \in IF$ *of size* $N \geq 1$ *(multiple interface inheritance for* $N \geq 2$ *allowed),*

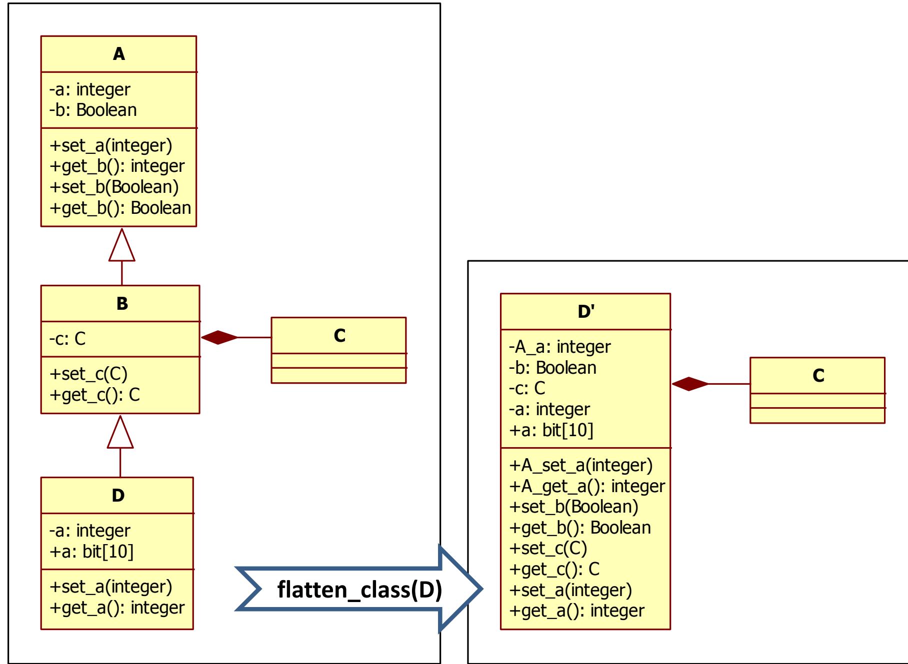


Figure 5.4: Example of *flatten_class* function applied to a class D. The flat class $D' = flatten_class(D)$ without inheritance and a flat/merged state vector and method list. Note that for conflicting attribute and methods the *classname_prefix*, in this example `A_`, has been applied.

2. *and a set (which can be empty) of interface member functions* $Method = \emptyset \cup \{im_0, \ldots im_m\}$ *with* im_i, $i \geq 0$ *of the following kinds:*

$$\begin{aligned}
&\textbf{\textit{virtual}}\ name\colon \textbf{\textit{void}} \to \textbf{\textit{void}} = \mathbf{0} && \textit{no arguments and no return type}\\
&\textbf{\textit{virtual}}\ name\colon \textbf{\textit{void}} \to T = \mathbf{0} && \textit{no arguments and return type } T\\
&\textbf{\textit{virtual}}\ name\colon T_0 \times \cdots \times T_n \to \textbf{\textit{void}} = \mathbf{0} && \textit{arguments } T_i,\ i \geq 0 \textit{ and no return type}\\
&\textbf{\textit{virtual}}\ name\colon T_0 \times \cdots \times T_n \to T = \mathbf{0} && \textit{arguments } T_i,\ i \geq 0 \textit{ and return type } T
\end{aligned}$$

Interface classes have no internal state. Interface methods are pure method prototypes (denoted by $= \mathbf{0}$*) without any implementation.*

Interface classes can be of the kinds in, out, and inout given by the following function:

$$kind(X) : IF \to \{in, out, inout\}$$

$$kind(X) = \begin{cases}
in, & \textit{if } \forall m \in X.Method:\\
& \quad m.kind = \textbf{\textit{void}} \to \textbf{\textit{void}} \ \vee\\
& \quad m.kind = T_0 \times \cdots \times T_n \to \textbf{\textit{void}},\\
out, & \textit{if } \forall m \in X.Method:\\
& \quad m.kind = \textbf{\textit{void}} \to T,\\
inout & \textit{if } \forall m \in X.Method:\\
& \quad m.kind = \textbf{\textit{void}} \to \textbf{\textit{void}} \ \vee\\
& \quad m.kind = \textbf{\textit{void}} \to T \ \vee\\
& \quad m.kind = T_0 \times \cdots \times T_n \to \textbf{\textit{void}} \ \vee\\
& \quad m.kind = T_0 \times \cdots \times T_n \to T.
\end{cases}$$

□

Also interface classes can normalized or flattened. The main difference from regular classes is that interface classes are not allowed to have a state, and thus no state vector flattening needs to be performed. Regarding the flattening of interface methods we need to consider the special properties:

- multiple inheritance of other interface classes is allowed,
- interface methods are pure prototypes, i.e. two interface methods are equal if they have the same name and the same signature.

With these properties, the normalized set of interface methods for a given list of base classes $\overline{c}_{parent} \in IF$, where $|\overline{c}_{parent}|$ is the number of elements of $\overline{c}_{parent}$, can be computed using the recursive function $flat_if(\overline{c}_{parent})$:

$$
\begin{aligned}
flat_if(X) &: \quad IF \rightarrow IF \\
flat_if(X) &= \quad X.Method, \quad \text{if } |X.\overline{c}_{parent}| = 0 \\
flat_if(X) &= \quad \left(\bigcup_{i=0}^{|X.\overline{c}_{parent}|-1} flat_if(c_i) \right) \cup X.Method, \ c_i \in X.\overline{c}_{parent}, \quad \text{if } |X.\overline{c}_{parent}| > 0
\end{aligned}
$$

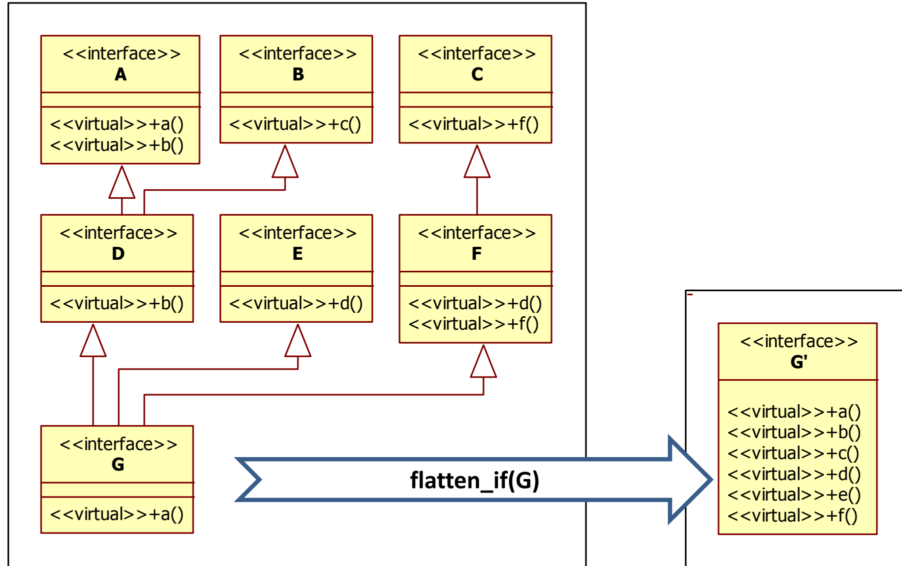


Figure 5.5: Example of $flatten_if$ function applied to a hierarchical interface class G. The flat interface class $G' = flatten_if(G)$ contains once each pure virtual method inherited by G.

5.4 Behavioral Layer

The proposed methodology starts with capturing the functional (or computational) behavior of the intended system to be designed. It shall be used to capture the computational behavior of the system in a way that it enables measuring the complexity of an algorithm in execution time and memory space. For the design of embedded real-time and resource constrained systems this is a very important prerequisite for supporting the decision process of mapping and implementation towards the embedded target platform.

5.4.1 Introduction

For capturing behavior our methodology shall support the following basic elements and composition rules for the description of a composable model of the application/system under design. The

presented model is inspired by SpecC [150, 160]. The main idea behind the composable behavior model is the explicit description of an applications structural granularity and dependencies. These information, only an application expert knows, shall be captured in this initial model. Subsequent refinement steps can exploit this expert information for the implementation on a resource constrained hardware/software platform.

The basic orthogonal design elements for capturing behavior are:

Data Types are the fundamentals of computation since instances of data types represent the objects or operands to be manipulated by corresponding operations. Algorithms are using operations on instances of data types to implement data transformation that represents the behavior.

Sequential Behavior describes control and data flow for the manipulation of data in a sequential order. It is the foundation of sequential imperative and object-oriented programming languages and the most intuitive way of describing algorithms. A language independent representation is a Control Data Flow Graph (CDFG).

Parallel Composition of Sequential Behavior allows the specification of application specific coarse grained concurrency. Parallel composition allows to specify multiple parallel threads of execution.

Synchronization for managing parallel composition of sequential behaviors. Synchronization is used to constrain the raw parallelism to guarantee consistency of data shared between parallel threads. Reliable communication between parallel threads relies on a proper synchronization scheme. An example of synchronization is the exchange of data (communication) between parallel threads using double- or single handshake protocol. Some examples of synchronization are pipelined (overlapping) threads or temporal parallelism following a fork-join semantics.

Hierarchical Composition is completely orthogonal to all design elements mentioned above, since it can be applied to all of them in the following way:

- **data types** can be hierarchical to express the composition of high-level data structures of low-level basic data types. In object-oriented programming languages the hierarchy can be expressed through aggregation, composition or inheritance. Where the relationship from aggregation to inheritance becomes stronger. Aggregate [UML Aggregates] data type can also exist without being part of a composition, the existence of composite [UML composition] data types depends on the existence of the data type that instantiated it, while inheritance [UML generalization] is a specialization of an existing data type which can only add features to its base type.
- **sequential behavior** can be structured by using functions that encapsulate sequential behavior which is used several times with possibly different parameters (e.g. functional programming completely relies on the composition of side-effect free (higher-order) functions). In object-oriented programming languages sequential behavior can also be hierarchically structured using objects which encapsulate data and corresponding operations/manipulations on these data. Furthermore, the concept of (hierarchical) finite-state machines can be used to describe hierarchical composition for sequential behaviors.
- **parallel composition** can also be hierarchical. Concurrent threads can contain more parallel compositions, e.g. a pipeline where each pipeline stage contains its own sup-pipeline or even more parallel sub threads. Or a parallel thread that uses join-fork and temporally spawns other parallel threads. But anyhow, hierarchical parallel composition needs to be handled with care since synchronization across hierarchies might become difficult to handle. Design languages supporting this kind of hierarchical parallel composition either embed this concept into the overall execution semantics of the model or offer special synchronization primitives for building and connection parallel compositions.

- **synchronization** is the key for managing parallelism and shall be separated from the pure algorithmic and computational parts. The separation between computation and communication is the main design concept to structure and also to use hierarchy in synchronization. The concept of communication channel refinement and the support of hierarchical channels for the description of complex communication protocol stacks allows hierarchical composition for synchronization and communication.

5.4.2 Modeling Elements

The Behavioral Layer is an executable parallel object-oriented model and consists of the following modeling elements:

5.4.2.1 Port

Definition 5.4.2.1 (Port):
A Port is a Tuple $Port = [Required_{IF}, Bound_{IF}]$ *with*

1. *a Required Interface Class* $Required_{IF} \in IF$. *The required interface contains all Services* S *as defined in the Methods of* $flat_if(Required_{IF})$. *All Services* $S \in Required_{IF}$ *are accessible through this port.*
2. *a Bound Interface* $Bound_{IF}$ *that provides or forwards the required interface* $Required_{IF}$. $Bound_{IF}$ *is either another port with the same Required_IF (hierarchical port to port binding, or forwarding) or a communication Channel (see Section 5.4.2.3) that implements the required interface. In this sense:*

$$Bound_{IF} \in \textbf{Channel} \cup \textbf{Port}$$

with the set of all communication Channels **Channel** *and the set of all Ports* **Port**.

The binding relation $\triangleright$ *describes Port to Port and Port to Interface bindings in the following way:*

$$\begin{aligned} \triangleright \quad &: \quad Port \times Port \\ p_i \triangleright p_j \quad &= \quad p_i.Bound_{IF} := p_j \text{ if } p_i.Required_{IF} \equiv p_j.Required_{IF} \wedge i \neq j \\ \triangleright \quad &: \quad Port \times Channel \\ p \triangleright ch \quad &= \quad p.Bound_{IF} := ch \text{ if } inh_closure(ch) \cap p.Required_{IF} \neq \emptyset \end{aligned}$$

For the sake of simplicity we define a function that eliminates port to port bindings as a recursive function. The function takes a Port as input and returns a flattened port that is directly connected to bound communication Channels.

$$\begin{aligned} flatten_binding(X) \quad &: \quad Port \rightarrow Port \\ flatten_binding(X) \quad &= \quad X \quad \text{if } X.Bound_{IF} \in \textbf{Channel} \\ flatten_binding(X) \quad &= \quad flatten_binding(X.Bound_{IF}) \quad \text{if } X.Bound_{IF} \in \textbf{Port} \end{aligned}$$

With this flattening we can define a valid port to interface binding as:

$$flatten_binding(Port).Bound_{IF} = Channel \Leftrightarrow Channel \models Port.Required_{IF}$$

In other words, the binding of a Port to a Channel is valid, if the Channel models or implements the required interface of the port. The relation "$\models$*" can be expressed as:*

$$Channel \models Port.Required_{IF} \Leftrightarrow inheritance_closure(Channel) \cap Port.Required_{IF} \neq \emptyset$$

The binding relation $\triangleright$ *can be flattened, denoted as* $\triangleright\downarrow_{Channel}$ *by applying* $flatten_binding$ *in the following way:*

$$\triangleright\downarrow_{Channel} \quad : \quad Port \times Port$$

$$p_i \rhd\!\downarrow_{Channel} p_j \;=\; p_i \rhd flatten_binding(p_j)$$
$$\rhd\!\downarrow_{Channel} \;:\; Port \times Channel$$
$$p \rhd\!\downarrow_{Channel} ch \;=\; p \rhd ch$$

Ports have the kinds $in, out,$ *or* $inout$ *of their required interface class, given by the following function:*

$$kind(X) \;:\; Port \rightarrow \{in, out, inout\}$$
$$kind(X) \;=\; kind(X.Required_{IF})$$

□

5.4.2.2 Behavior

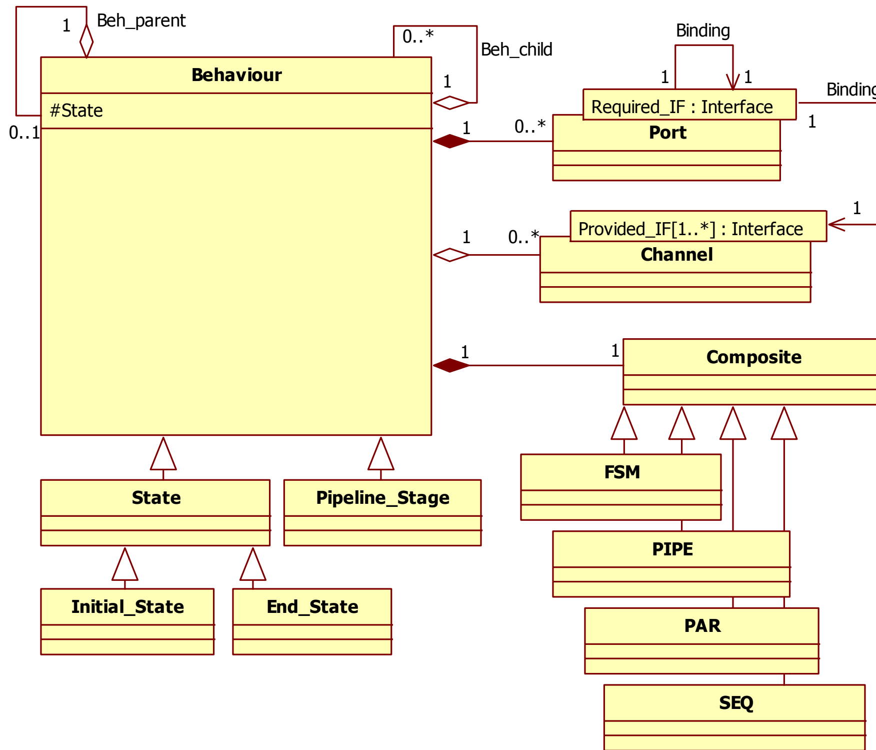


Figure 5.6: Meta-Model of Behavior

Definition 5.4.2.2 (Behavior):
Behavior encapsulates the system's functionality (in Leaf Behaviors), and specifies the causality of the system through explicit scheduling using hierarchical and composite (sequential, finite state machine, parallel or pipelined) behaviors. Communication is explicitly separated from functionality and encapsulated in Channels (see Section 5.4.2.3). The behavior in this work describes a reduced kind of Program State Machine (PSM), as introduced in Section 3.6.2. Timing itself is not part of the Behavior, but can be annotated inside Leaf Behaviors.

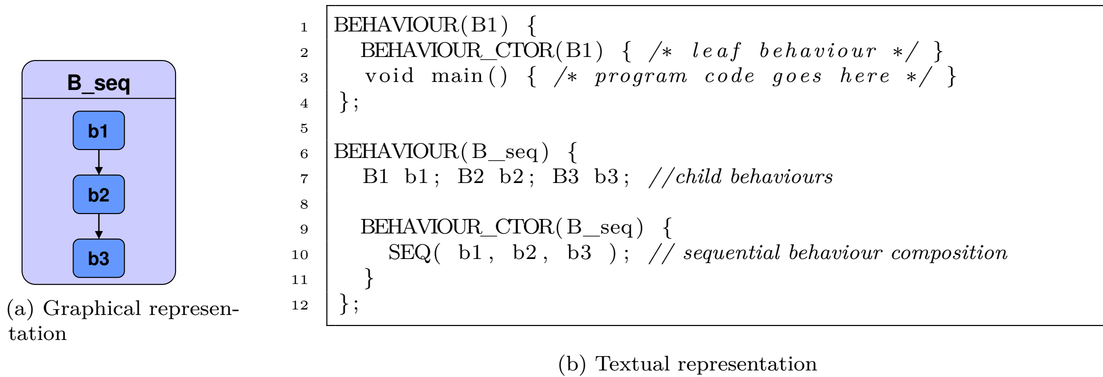
A Behavior is defined as a tuple

$$Behavior = \left[Type, Composite, Beh_{parent}, \overline{Beh_{Child}}, State, \overline{Port}, \overline{Channel}, \overline{Binding}\right],$$

and the corresponding UML Meta-Model is shown in Figure 5.6:

1. $Type = \{regular, state, initial_state, end_state, pipeline_stage\}$ *is the type of the Behavior, with the following semantics:*

   $regular$ *is a regular Behavior and can either be the root Behavior or is part of a either a sequential (SEQ) or parallel (PAR) composition.*



(a) Graphical representation

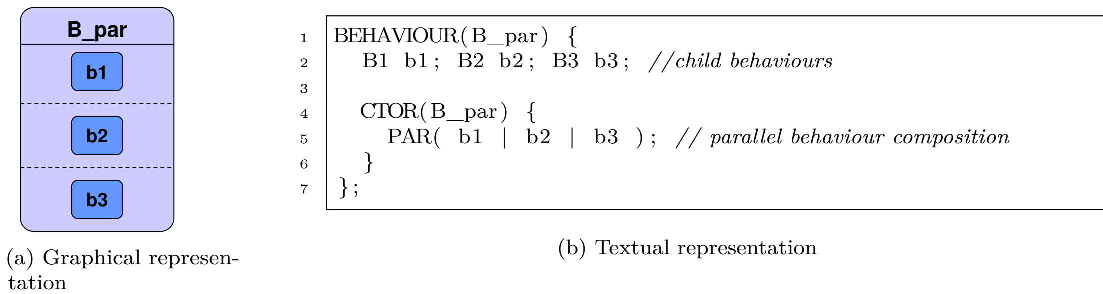
```
BEHAVIOUR(B1) {
  BEHAVIOUR_CTOR(B1) { /* leaf behaviour */ }
  void main() { /* program code goes here */ }
};

BEHAVIOUR(B_seq) {
  B1 b1; B2 b2; B3 b3; //child behaviours

  BEHAVIOUR_CTOR(B_seq) {
    SEQ( b1, b2, b3 ); // sequential behaviour composition
  }
};
```

(b) Textual representation

Figure 5.7: Example of Sequential (SEQ) composite behavior



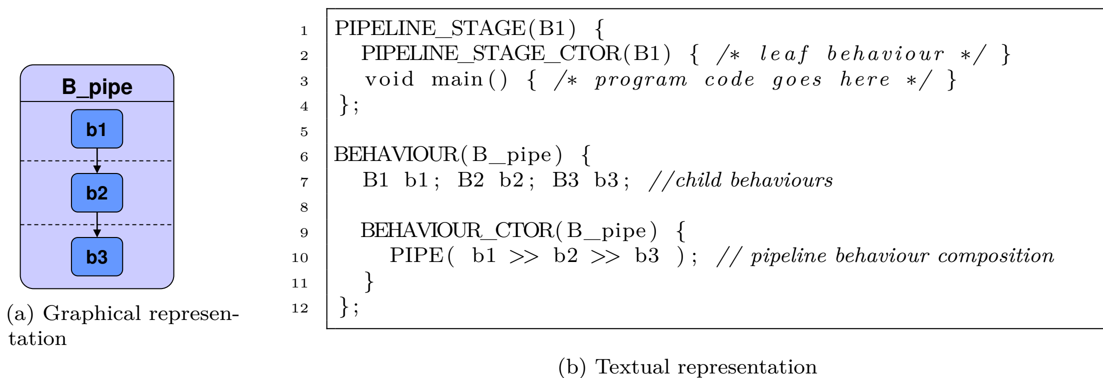
(a) Graphical representation

```
BEHAVIOUR(B_par) {
  B1 b1; B2 b2; B3 b3; //child behaviours

  CTOR(B_par) {
    PAR( b1 | b2 | b3 ); // parallel behaviour composition
  }
};
```

(b) Textual representation

Figure 5.8: Example of Parallel (PAR) composite behavior



(a) Graphical representation

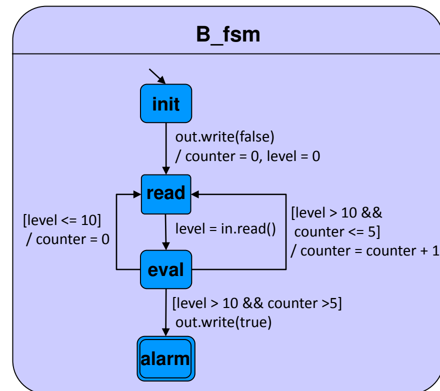
```
PIPELINE_STAGE(B1) {
  PIPELINE_STAGE_CTOR(B1) { /* leaf behaviour */ }
  void main() { /* program code goes here */ }
};

BEHAVIOUR(B_pipe) {
  B1 b1; B2 b2; B3 b3; //child behaviours

  BEHAVIOUR_CTOR(B_pipe) {
    PIPE( b1 >> b2 >> b3 ); // pipeline behaviour composition
  }
};
```

(b) Textual representation

Figure 5.9: Example of Pipeline ($PIPE$) composite behavior

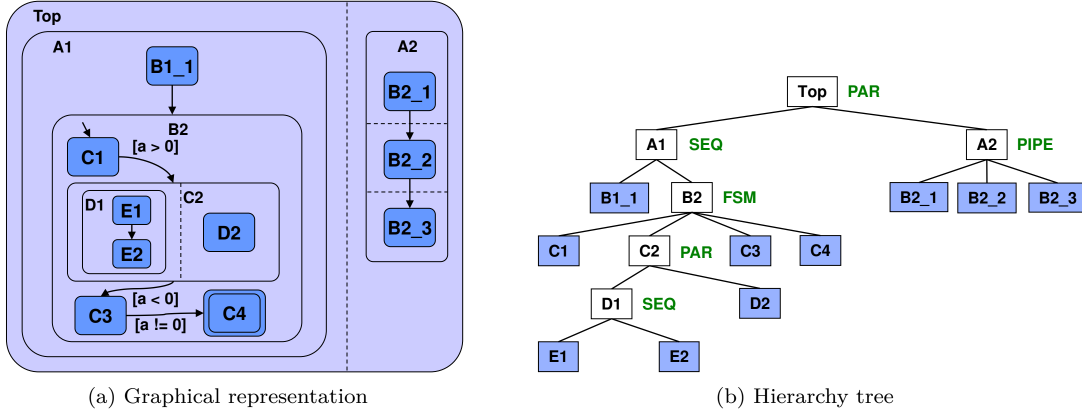


(a) Graphical representation

```
1  BEHAVIOUR(B_fsm) {
2    Port_in<integer>  in;
3    Port_out<Boolean> out;
4    integer level;
5    integer counter;
6
7    Start_State init;
8    State read, eval;
9    End_State alarm;
10
11   BEHAVIOUR_CTOR(B_fsm) {
12     // finite state-machine child behaviour composition
13     FSM(
14       (init >> read, GUARD(true), CALL(B_fsm::c1), UPDATE(B_FSM::u1)) &&
15       (read >> eval, GUARD(true), CALL(B_fsm::c2), UPDATE()) &&
16       (eval >> read, GUARD(B_fsm::g3), CALL(), UPDATE(B_FSM::u3)) &&
17       (eval >> read, GUARD(B_fsm::g4), CALL(), UPDATE(B_FSM::u4)) &&
18       (eval >> alarm, GUARD(B_fsm::g5), CALL(B_fsm::c5), UPDATE())
19     );
20   }
21
22   // guard functions (no side-effect)
23   bool g3() const { return (level <= 10); }
24   bool g4() const { return ((level > 10) && (counter <= 5)); }
25   bool g5() const { return ((level > 10) && (counter > 5)); }
26
27   // call functions (with internal and external side-effect)
28   void c1() { out.write(false); }
29   void c2() { level = in.read(); }
30   void c5() { out.write(true); }
31
32   // update functions (with internal side-effect only)
33   void u1() { counter = 0; level = 0; }
34   void u3() { counter = 0; }
35   void u4() { counter = counter + 1; }
36 };
```

(b) Textual representation

Figure 5.10: Example of Finite State Machine (FSM) composite behavior



(a) Graphical representation (b) Hierarchy tree

Figure 5.11: Example of a complex hierarchical composition of Behaviors. `Top` is a parallel composite behavior, `A1` is a sequential, and `A2` is a pipelined composite behavior. `B2` is a finite state machine, `C2` a parallel, and `D1` a sequential composition. Leaf behaviors `B1_1`, `B2_1`, `B2_2`, `B2_3`, `C1`, `C3`, `C4`, `D2`, `E1`, and `E2` may contain program code.

state *is the basic element of a finite state machine (FSM) composition.*

initial_state *is a special state type that marks the initial or start state of a FSM composition. A FSM can only have a single start state.*

end_state *is a special state type that marks the final or end state of a FSM composition. A FSM can have 1 to N, with $N \leq number_of_states - 1$ end states.*

pipeline_stage *is the basic element of a pipeline (PIPE) composition.*

2. *$Composite = \{SEQ, FSM, PAR, PIPE\} \cup \emptyset$ describes the composite type of the Behavior. A composite behavior describes the execution order of the Child Behaviors.*

Leaf *describes a leaf behavior when $Composite = \emptyset \wedge \overline{Beh}_{Child} = \emptyset$. Leaf behaviors contain pure computation described as Control Data Flow Graph (CDGF) of Flow Graph. A leaf behavior is executed in a run-to-completion fashion. Once started it executes from the root to one of the leaf nodes of the of the associated CDFG. The activation of a leaf behavior depends on the execution order as defined by its patent Behavior Beh_{parent}.*

Sequential (SEQ) *composite type requires at least 2 child behaviors $|\overline{Beh}_{Child}| \geq 2$ with $\forall B \in \overline{Beh}_{Child} : B.Type = regular$. With $\{b_0, \cdots, b_N\}$, $N \geq 1$ the SEQ-Constructor $SEQ(b_0, \cdots, b_N)$ defines the sequential execution sequence $b_0 \to \cdots \to b_N$. The execution of a SEQ composite behavior is completed when sub behavior b_N is completed. Figure 5.7 shows an example of a sequential behavior.*

Finite State Machine (FSM) *composite type requires at least 2 child behaviors $|\overline{Beh}_{Child}| \geq 2$ with*

$$\exists! B \in \overline{Beh}_{Child} : B.Type = start_state \wedge \exists B \in \overline{Beh}_{Child} : B.Type = end_state.$$

The finite state machine constructor is $FSM(S, P, V, F, H, s_0, E)$ with:

(a) $S = \{s_0, s_1, \cdots, s_l\}$ is the set of all states, with $l \geq 1$ number of states. $S \subseteq \overline{Beh}_{Child}$ and $\forall s \in S \setminus (s_0 \cup E)$: $s.Type = state$.

(b) $P = \{p_0, p_1, \cdots, p_m\}$ is a set of inputs, output and combined input and output ports, with $I \subseteq \overline{Port}$, with

$$\forall i \in I : kind(i) = in \vee kind(i) = out \vee kind(i) = inout.$$

(c) $V = \{v_0, v_1, \cdots, v_n\}$ is a set of variables, with $V \subseteq \overline{State}$.

(d) *$F\colon S \times G \times P \to S$ is a next-state function, that can be represented by a transition table. A transition t from state $s_i \in S$ to state $s_j \in S$ (where $i = j$ describes a transition from a state to itself, a so-called self-transition) can be taken when the side-effect free Boolean guard condition G on the set of variables $v_i \in V$ can be evaluated to true, and the interface method call on Port $p_k \in P$ returns. For taking a transition the guard evaluation function $G\colon V \to \textbf{Boolean}$ needs to evaluate to true. If the guard is false no interface method call will be issued. A transition t can be of the following kinds:*

$$s_i \xrightarrow{[G\colon V \to \textbf{Boolean}],\, v_n := p_k.method_l(\overline{v})} s_j \quad \textit{guarded synchronization (read/write)}$$

$$s_i \xrightarrow{v_n := p_k.method_l(\overline{v})} s_j \quad \textit{unguarded synchronization (read/write)}$$

$$s_i \xrightarrow{[G\colon V \to \textbf{Boolean}],\, p_k.method_l(\overline{v})} s_j \quad \textit{guarded synchronization (write)}$$

$$s_i \xrightarrow{p_k.method_l(\overline{v})} s_j \quad \textit{unguarded synchronization (write)}$$

$$s_i \xrightarrow{[G\colon V \to \textbf{Boolean}]} s_j \quad \textit{guarded transition}$$

$$s_i \longrightarrow s_j \quad \textit{unconditional/unguarded transition}$$

with $v_n \in V$, $method_l \in \textbf{IF}$, and $\overline{v} = \{v_0, \cdots v_N\} \cup \emptyset$ where $0 \leq N \leq |V| - 1$.

(e) *$H\colon F \times V \to V$ is an output function that describes a state and communication dependent update of the local Variables $V \subseteq \overline{State}$ between a transition from one state to another state. State updates either occur after read/write synchronization through interface method call of type out or inout via port P or through an explicit variable update after the guard condition and synchronization call have been successfully completed. The explicit variable update is a function $U\colon V \to V$ With H the transition t is extended with the following way:*

$$s_i \xrightarrow[/U\colon V \to V]{[G\colon V \to \textbf{Boolean}],\, v_n := p_k.method_l(\overline{v})} s_j \quad \textit{guarded synchronization (read/write)}$$

$$s_i \xrightarrow[/U\colon V \to V]{v_n := p_k.method_l(\overline{v})} s_j \quad \textit{unguarded synchronization (read/write)}$$

$$s_i \xrightarrow[/U\colon V \to V]{[G\colon V \to \textbf{Boolean}],\, p_k.method_l(\overline{v})} s_j \quad \textit{guarded synchronization (write)}$$

$$s_i \xrightarrow[/U\colon V \to V]{p_k.method_l(\overline{v})} s_j \quad \textit{unguarded synchronization (write)}$$

$$s_i \xrightarrow[/U\colon V \to V]{[G\colon V \to \textbf{Boolean}]} s_j \quad \textit{guarded transition}$$

$$s_i \xrightarrow[/U\colon V \to V]{} s_j \quad \textit{unconditional/unguarded transition}$$

(f) *$s_0 \in S$ is the only initial state, with $s_0.Type = initial_state$.*

(g) *$E = \{e_0, e_1, \cdots, e_p\} \subseteq S$ is a set of end states, with $\forall e \in E\colon e.Type = end_state$. When an end state is entered the FSM behavior is finished. An end state has no sub Behavior $\forall e \in E\colon |e.\overline{Beh}_{Child}| = 0$.*

Figure 5.10 shows an example of a Finite State Machine composite behavior.

***Parallel** (PAR) composite type requires at least 2 child behaviors $|\overline{Beh}_{Child}| \geq 2$ with $\forall B \in \overline{Beh}_{Child} : B.Type = regular$. With $\{b_0, \cdots, b_N\}$, $N \geq 1$ the PAR-Constructor $PAR(b_0 | \cdots | b_N)$ defines the parallel execution $b_0 || \cdots || b_N$. The execution of a PAR composite behavior is completed when all sub behaviors $b_0, \cdots, b_N$ are completed. Figure 5.8 shows an example of a parallel composite behavior.*

***Pipeline** ($PIPE$) composite type requires at least 2 child behaviors $|\overline{Beh}_{Child}| \geq 2$ with $\forall B \in \overline{Beh}_{Child} : B.Type = pipeline_stage$. With $\{b_0, \cdots, b_N\}$, $N \geq 1$ the PIPE-Constructor $PIPE(b_0 >> \cdots >> b_N)$ defines the pipelined execution sequence $b_0 >> \cdots >> b_N$. If a pipelined execution is not constrained it executes in an infinite loop. If a pipeline execution is constrained by M $PIPE(b_0 >> \cdots >> b_N)_M$, with*

$M \geq N-1$, each pipeline state is executed M times and the pipeline behavior is completed.
For $N = 3$ the unconstrained pipeline execution

$$PIPE(b_0 >> b_1 >> b_2 >> b_3)_\infty$$

is:

$$\begin{array}{cccccccc}
 & & & & & & & \dots \\
 & & & & & & b_0 & \dots \\
 & & & & & b_0 & b_1 & \dots \\
 & & & & b_0 & b_1 & b_2 & \dots \\
 & & & b_0 & b_1 & b_2 & b_3 & \\
 & & b_0 & b_1 & b_2 & b_3 & & \\
 & b_0 & b_1 & b_2 & b_3 & & & \\
b_0 & b_1 & b_2 & b_3 & & & &
\end{array}$$

For $N = 3$ and $M = 3$ the constrained pipeline execution

$$PIPE(b_0 >> b_1 >> b_2 >> b_3)_3$$

is:

$$\begin{array}{cccccc}
 & & b_0 & b_1 & b_2 & b_3 \\
 & b_0 & b_1 & b_2 & b_3 & \\
b_0 & b_1 & b_2 & b_3 & &
\end{array}$$

Figure 5.9 shows an example of a pipelined composite behavior.

3. *$Beh_{parent} \in$ **Behavior** $\cup \emptyset$ is the parent Behavior. If $Beh_{parent} = \emptyset$ we call the Behavior the Root Behavior.*
4. *$\overline{Beh}_{Child} = \{b_0, \cdots, b_N\}$ is a set of child or sub Behaviors. If $\overline{Beh}_{Child} = \emptyset$ we call this Behavior a Leaf Behavior.*
5. *$State = T_0 \times \cdots \times T_n$, with $n \geq 0$ where $T_0, \dots T_n$ denote some basic types, classes, or arrays is the state vector of the Behavior. Only Leaf Behavior are allowed to have a non-empty state: $\forall B \in$ **Behavior**: $B.\overline{Beh}_{Child} \neq \emptyset \Rightarrow B.State = \emptyset$. Figure 5.11 shows an example of a complex behavior hierarchy. The behavior tree shows the Beh_{parent} and $\overline{Beh}_{Child}$ relation along the hierarchy.*
6. *$\overline{Port} = \{p_0, \cdots, p_N\}$ is a set of Ports with $p_i \in$ **Port**, $0 \leq i \leq N$ and $N \geq 0$. Note that $\overline{Port} \neq \emptyset$ since ports are the only way to communicate with child or parent behaviors. Ports have strict binding rules. In general multiple ports can be bound to the same interface of a Channel, but hierarchical port to port binding is exclusive. I.e. it is not allowed to bind a port to two different ports in a child behavior. In other words, the intersecting set of end ports (directly bound to a channel's interface, and obtained using the $flatten_binding$ function on a port) in the binding path from ports of all leaf behaviors is empty:*

$$\bigcap_{i=0}^{|leaf_beh(X)|-1} \left(\bigcap_{j=0}^{|b_i.\overline{Port}|-1} b_i.flatten_binding(p_j) \right) = \emptyset$$

*with $X \in$ **Behavior**, $b_i \in leaf_beh(X)$, and $p_j \in b_i.\overline{Port}$.*
*The set of leaf Behaviors from a start Behavior $X \in$ **Behavior** can be computed:*

$$\begin{array}{lcl}
leaf_beh(X) & : & Behavior \rightarrow \{Behavior_0, \cdots, Behavior_N\} \\
leaf_beh(X) & = & X \quad if\ |X.\overline{Beh}_{Child}| = 0 \\
leaf_beh(X) & = & \displaystyle\bigcup_{i=0}^{|X.\overline{Beh}_{Child}|-1} leaf_beh(x_i),\ x_i \in \overline{Beh}_{Child} \quad if\ |X.\overline{Beh}_{Child}| > 0
\end{array}$$

7. *$\overline{Channel} = \{ch_0, \cdots, ch_N\} \cup \emptyset$ is the set of Channels with $N \geq 0$ and $ch_i \in$ **SharedVariable** $\cup$ **PipedVariable** $\cup$ **Queue** $\cup$ **Handshake** $\cup$ **DoubleHandshake** for $0 \leq i \leq N$.*

8. $\overline{Binding} = \{\rhd_0, \cdots, \rhd_N\}$ *is a list of binding relations* $\rhd$ *with*

$$N = \left(\sum_{i=0}^{|\overline{Beh}_{child}|-1} |b_i.\overline{Port}| \right) - 1$$

where $b_i \in \overline{Beh}_{child}$*. The list of binding relations has the following properties:*

(a) *Ports of all child Behaviors are bound once to either Ports of the Barbour or interfaces of Channels in the Behavior:*

$$\forall_{b_i \in \overline{Beh}_{child}} \forall_{p_j \in b_i.\overline{Port}} : \exists! \rhd \in Binding \text{ with } p_j \rhd p \in \overline{Port} \vee p_j \rhd ch \in \overline{Channel}$$

(b) *One port is uniquely bound to another port. I.e. not two different ports can be bound to the same port:*

$$\forall_{\rhd_i \in Binding} \quad p \in \bigcup_{b \in \overline{Beh}_{child}} b.\overline{Port} : \exists! p_j \in \overline{Port} \wedge p \neq p_j \text{ with } p \rhd_i p_j$$

□

Definition 5.4.2.3 (Parallel Set):
A Parallel Set of a Behavior B consists of pure Sequential Sets of all child behaviors of B. A Sequential Set only contains composite behaviors of type SEQ and FSM. The parallel set is defined as

$$\begin{aligned}
par_set(X) &: Behavior \rightarrow \wp(Behavior) \\
par_set(X) &= append_to_set(X) \quad \text{if } |X.\overline{Beh}_{Child}| = 0 \\
par_set(X) &= \begin{cases} \forall x_i \in \overline{Beh}_{Child} : (new_set(); \\ \qquad par_set(x_i); \\ \qquad close_set()) & \text{if } (X.Composite = PAR \vee \\ & \quad X.Composite = PIPE) \wedge \\ & \quad |X.\overline{Beh}_{Child}| > 0 \\ \forall x_i \in \overline{Beh}_{Child} : par_set(x_i) & \text{if } X.Composite \neq PAR \wedge \\ & \quad X.Composite \neq PIPE \wedge \\ & \quad |X.\overline{Beh}_{Child}| > 0 \end{cases}
\end{aligned}$$

with

$$\begin{aligned}
new_set() &: \text{creates new set}, \\
append_to_set(X) &: \text{appends element } X \text{ to last created set}, \\
close_set() &: \text{closes last created set}.
\end{aligned}$$

The number of parallel sets in $\wp(Behavior)$ *is defined as* $|par_set(X)|$, $X \in$ **Behavior***. Figure 5.12 gives an example on the computation of par_set. The number of parallel sets describes the amount of potentially concurrently running running behavior clusters. For a Sequential Set* $|par_set(X)| = 1$. □

5.4.2.3 Channels

For the communication between leaf behaviors a set of pre-defined channels is provided. Figure 5.13 gives an overview of the Behavioral Layer channels. In the following, a definition and property description of the different channel types is given. This subsection closes with a methodical restriction on the usage of these channel types in the Behavioral Layer models.

Definition 5.4.2.4 (Channel):
A Channel is a tuple $CH = [IF, Service]$ *with*

1. $IF \in$ **Interface** *is the set of interfaces this channel provides.*

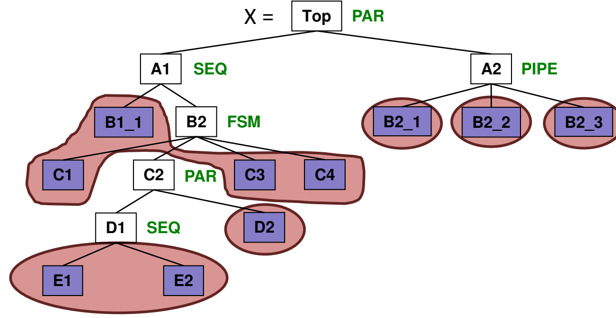


Figure 5.12: Example: Parallel Set computed on hierarchy tree of example from Figure 5.11. $par_set(X) = \{\{B1_1, C1, C3, C4\}, \{E1, E2\}, \{D2\}, \{B2_1\}, \{B2_2\}, \{B2_3\}\}$, $|par_set(X)| = 6$.

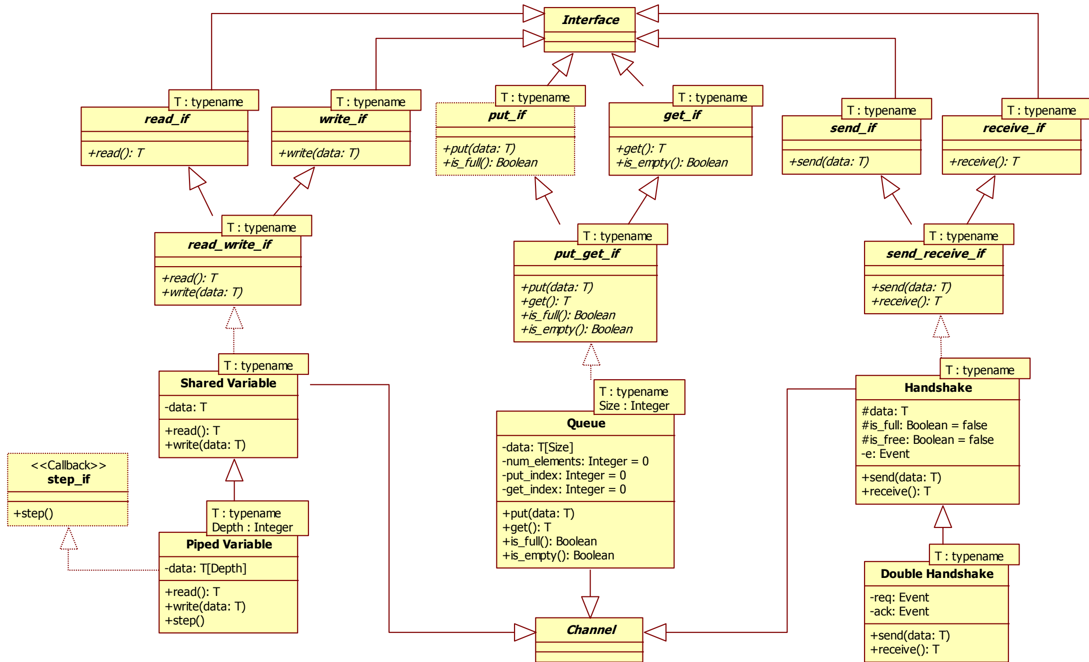


Figure 5.13: Channels of the Behavioral Layer

2. *Service* $\models$ *IF is the set of all service implementations of this channel as required by the set of provided interfaces IF.*

□

Definition 5.4.2.5 (Shared Variable):
A Shared Variable is a tuple $SV = [Type, IF, State, Service]$ *and a special Channel* $CH = [IF, Service]$ *with the following properties:*

1. $Type \in \textbf{\textit{integer}}[N] \cup \textbf{\textit{Boolean}}[N] \cup \textbf{\textit{bit}}[N] \cup \textbf{\textit{Class}}$ *with* $N \geq 0$*, where* $N = 0$ *is a shared variable of a basic type or class and* $N > 0$ *is a shared variable of an array of a basic type of size* $N + 1$*.*

2. $IF = \{read_if\langle Type\rangle, write_if\langle Type\rangle, read_write_if\langle Type\rangle\}$ *with* $read_if\langle Type\rangle \in \textbf{Interface}$

$$\textbf{\textit{virtual}}\ read: \textbf{\textit{void}} \rightarrow Type = \mathbf{0}$$

and $write_if\langle Type\rangle \in \textbf{Interface}$

$$\textbf{\textit{virtual}}\ write: Type \rightarrow \textbf{\textit{void}} = \mathbf{0}$$

and read_write_if<Type> $\in$ **Interface**

$$\begin{aligned} \textbf{\textit{virtual}}\ read &: \textbf{\textit{void}} \rightarrow Type = \mathbf{0} \\ \textbf{\textit{virtual}}\ write &: Type \rightarrow \textbf{\textit{void}} = \mathbf{0} \end{aligned}$$

3. *State* $= \{data\}$ *with* $data \in Type$

4. *Service* $= \{read, write\}$ *of the following kinds:*

$$\begin{aligned} read &: \textbf{\textit{void}} \rightarrow Type \\ read() &= \{\textbf{\textit{return}}\ data\} \\ write &: Type \rightarrow \textbf{\textit{void}} \\ write(in) &= \{data = in\} \end{aligned}$$

□

Properties of the Shared Variable, as defined in the SpecC reference implementation [229]:

- one channel instance is required for each shared variable
- multiple leaf behaviors may use the same shared variable instance
- each connected leaf behavior may act as a receiver or sender or both (depending on the used interface)
- a sender calls `write()` to store data into the shared variable
- a receiver calls `read()` to retrieve data from the shared variable
- when using the shared variable with multiple parallel or piped behaviors no guarantees are given for fairness of access, improper usage can cause race conditions and data inconsistencies

Definition 5.4.2.6 (Piped Variable):
A Piped Variable is a tuple $PV = [Type, Depth, IF, State, Service, Callback]$ *and a special Shared Variable* $SV = [Type, IF, State, Service]$ *with the following properties:*

1. $Type = SV.Type$*, same type constraints as Shared Variable.*

2. $Depth \in \textbf{\textit{integer}}_{[0,\infty]}$ *is the depth of the pipelined variable. It defines the number of pipeline steps after the the written data can be accessed via read service.*

3. $IF = SV.IF$*, same interface as Shared Variable.*

4. *State* $= \{data\}$ *with* $data[Depth + 1] \in Type$

5. *Service* $= \{read, write\}$ *of the following kinds:*

$$\begin{aligned} read &: \textbf{\textit{void}} \rightarrow Type \\ read() &= \{\textbf{\textit{return}}\ data[Depth]\} \\ write &: Type \rightarrow \textbf{\textit{void}} \\ write(in) &= \{data[0] = in\} \end{aligned}$$

6. *Callback of the following kind:*

$$\begin{aligned} step &: \textbf{\textit{void}} \rightarrow \textbf{\textit{void}} \\ step() &= \{\forall i \in \{0, \ldots, Depth - 1\} : data[Depth - i)] = data[Depth - (i + 1)]\} \end{aligned}$$

The step function is triggered by the pipeline controller (see Figure 5.33) after the completion of each pipeline cycle. The function shifts all elements in the data array by one position (from lower to higher index).

□

Properties of the Piped Variable, as defined in the SpecC reference implementation [229]:

- piped variables can only be used in Pipeline composite behaviors
- one channel instance is required for each piped variable
- multiple pipeline stages may use the same piped variable instance
- each connected pipeline stage may act as a receiver or sender or both (depending on the used interface)
- a sender calls `write()` to store data into the piped variable
- a receiver calls `read()` to retrieve data from the piped variable
- when using the piped variable with multiple piped behaviors no guarantees are given for fairness of access, improper usage can cause race conditions and data inconsistencies

Definition 5.4.2.7 (Queue):
A Queue is a tuple $Q = [Type, Size, IF, State, Service]$ *and a special Channel* $CH = [IF, Service]$ *with the following properties:*

1. $Type \in \textbf{\textit{integer}}[N] \ \cup \ \textbf{\textit{Boolean}}[N] \ \cup \ \textbf{\textit{bit}}[N] \ \cup \textbf{\textit{Class}}$ *with* $N \geq 0$*, where* $N = 0$ *is a shared variable of a basic type or class and* $N > 0$ *is a queue of an array of a basic type of size* $N + 1$*.*

2. $Size \in \textbf{\textit{integer}}_{>0}$ *is the size (i.e. number of elements) of this queue.*

3. $IF = \{put_if{<}Type{>}, get_if{<}Type{>}, put_get_if{<}Type{>}\}$ *with* $put_if{<}Type{>} \in$ **Interface**

$$\begin{aligned} \textbf{\textit{virtual}}\ put &: Type \rightarrow \textbf{\textit{void}} = \mathbf{0} \\ \textbf{\textit{virtual}}\ is_full &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}} = \mathbf{0} \end{aligned}$$

and $get_if{<}Type{>} \in$ **Interface**

$$\begin{aligned} \textbf{\textit{virtual}}\ get &: \textbf{\textit{void}} \rightarrow Type = \mathbf{0} \\ \textbf{\textit{virtual}}\ is_empty &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}} = \mathbf{0} \end{aligned}$$

and $put_get_if{<}Type{>} \in$ **Interface**

$$\begin{aligned} \textbf{\textit{virtual}}\ put &: Type \rightarrow \textbf{\textit{void}} = \mathbf{0} \\ \textbf{\textit{virtual}}\ get &: \textbf{\textit{void}} \rightarrow Type = \mathbf{0} \\ \textbf{\textit{virtual}}\ is_full &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}} = \mathbf{0} \\ \textbf{\textit{virtual}}\ is_empty &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}} = \mathbf{0} \end{aligned}$$

4. $State = \{data[Size], num_elements = 0, put_index = 0, get_index = 0\}$ *with* $data[Size] \in Type[Size]$ *and* $num_elements, put_index, get_index \in \textbf{\textit{integer}}_{>0}$*. The state variables* $num_elements, put_index, get_index$ *are all initialised with zero.*

5. $Service = \{put, get, is_full, is_empty\}$ *of the following kinds:*

$$\begin{aligned} put &: Type \rightarrow \textbf{\textit{void}} \\ put(in) &= \{ \\ &\qquad data[put_index] = in \\ &\qquad \textbf{\textit{if}}\ (put_index == (Size - 1))\ \textbf{\textit{then}}\ put_index = 0 \\ &\qquad \textbf{\textit{else}}\ put_index = put_index + 1 \\ &\qquad num_elements = num_elements + 1 \\ &\ \ \} \\ get &: \textbf{\textit{void}} \rightarrow Type \\ get() &= \{ \\ &\qquad Type\, tmp = data[get_index] \\ &\qquad \textbf{\textit{if}}\ (get_index == (Size - 1))\ \textbf{\textit{then}}\ get_index = 0 \end{aligned}$$

$$\begin{aligned}
&\qquad\qquad \textbf{\textit{else}}\ get_index = get_index + 1\\
&\qquad\qquad num_elements = num_elements - 1\\
&\qquad\qquad \textbf{\textit{return}}\ tmp\\
&\qquad \}\\
is_full &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}}\\
is_full() &= \{\textbf{\textit{return}}\ num_elements == Size\}\\
is_empty &: \textbf{\textit{void}} \rightarrow \textbf{\textit{Boolean}}\\
is_empty() &= \{\textbf{\textit{return}}\ num_elements == 0\}
\end{aligned}$$

□

Properties of the Queue, as defined in the SpecC reference implementation [229]:

- the queue operates in first-in-first-out (FIFO) mode
- one channel instance is required for each queue
- multiple leaf behaviors may use the same channel instance
- each connected leaf behavior may act as a receiver or sender or both (depending on the used interface)
- a sender calls `put()` to store data into the queue
- a receiver calls `get()` to retrieve data from the queue
- the queue is organized as a ring buffer, if insufficient space is available in the queue, a call of `put()` will corrupt exiting data in the queue. Calling the `is_full()` function before calling `put()` is recommended to avoid data corruption
- before calling `get()` on the queue, the `is_empty()` function should be called to test whether the queue is empty or not
- when using this queue with multiple parallel or piped behaviors no guarantees are given for fairness of access, improper usage can cause race conditions and data inconsistencies

Definition 5.4.2.8 (Handshake):
A Handshake is a tuple $HS = [Type, IF, State, Event, Service]$ and a special Channel $CH = [IF, Service]$ with the following properties:

1. *$Type \in$ **void** $\cup$ **integer**$[N]$ $\cup$ **Boolean**$[N]$ $\cup$ **bit**$[N]$ $\cup$ **Class** with $N \geq 0$, where $N = 0$ is a handshake channel of a basic type or class and $N > 0$ is a handshake channel of an array of a basic type of size $N + 1$. If the type of the handshake channel is **void**, no data is transported between sender and receiver. In this case only handshake synchronization is performed.*

2. *$IF = \{send_if\text{<}Type\text{>}, receive_if\text{<}Type\text{>}, send_receive_if\text{<}Type\text{>}\}$ with $send_if\text{<}Type\text{>} \in$* **Interface**

$$\textbf{\textit{virtual}}\ send\colon Type \rightarrow \textbf{\textit{void}} = \mathbf{0}$$

and $receive_if\text{<}Type\text{>} \in$ **Interface**

$$\textbf{\textit{virtual}}\ receive\colon \textbf{\textit{void}} \rightarrow Type = \mathbf{0}$$

and $send_receive_if\text{<}Type\text{>} \in$ **Interface**

$$\begin{aligned}
\textbf{\textit{virtual}}\ send &: Type \rightarrow \textbf{\textit{void}} = \mathbf{0}\\
\textbf{\textit{virtual}}\ receive &: \textbf{\textit{void}} \rightarrow Type = \mathbf{0}
\end{aligned}$$

3. *$State = \{data, is_free = \textbf{\textit{false}}, is_waiting = \textbf{\textit{false}}\}$ with $data \in Type$ and $is_free, is_waiting \in \textbf{\textit{Boolean}}$*

4. *$Event = \{e\}$*

5. *Service* = $\{send, receive\}$ *of the following kinds:*

$$
\begin{aligned}
send &: Type \rightarrow \textbf{\textit{void}} \\
send(in) &= \{ \\
&\quad data = in \\
&\quad \textbf{\textit{if}}\ is_waiting == \textbf{\textit{true then notify}}(e) \\
&\quad is_free = \textbf{\textit{true}} \\
&\}
\\
receive &: \textbf{\textit{void}} \rightarrow Type \\
receive() &= \{ \\
&\quad \textbf{\textit{if}}\ is_free == \textbf{\textit{false then}} \\
&\quad\quad is_waiting = \textbf{\textit{true}} \\
&\quad\quad \textbf{\textit{wait}}(e) \\
&\quad\quad is_waiting = \textbf{\textit{false}} \\
&\quad is_full = \textbf{\textit{false}} \\
&\quad \textbf{\textit{return}}\ data \\
&\}
\end{aligned}
$$

□

Properties of the Handshake Channel, as defined in the SpecC reference implementation [229]:

- this handshake channel provides safe one-way synchronization between a sender and a receiver
- only one sender and one receiver may use the channel at any time; otherwise, the behavior is undefined
- a call to `send()` sends a handshake to the receiver; if the receiver is waiting at the time of the `send()`, it will wake up and resume its execution; otherwise, the handshake is stored until the receiver calls `receive()`
- the behavior is undefined if `send()` is called successively without any calls to `receive()`
- a call to `receive()` lets the receiver wait for a handshake from the sender
- if a handshake is present at the time of `receive()`, the call to `receive()` will immediately return
- if no handshake is present at the time of `receive()`, the receiver is suspended until the sender sends the handshake; then, the receiver will resume its execution
- calling `send()` will not suspend the sender
- calling `receive()` may suspend the receiver indefinitely

Definition 5.4.2.9 (Double Handshake):
A Double Handshake is a tuple $DHS = [Type, IF, State, Event, Service]$ *and a special Handshake Channel* $HS = [Type, IF, State, Event, Service]$ *with the following properties:*

1. $Type = HS.Type$
2. $IF = HS.IF$
3. $State = HS.State$
4. $Event = \{req, ack\}$
5. *Service* = $\{send, receive\}$ *of the following kinds:*

$$
\begin{aligned}
send &: Type \rightarrow \textbf{\textit{void}} \\
send(in) &= \{ \\
&\quad data = in
\end{aligned}
$$

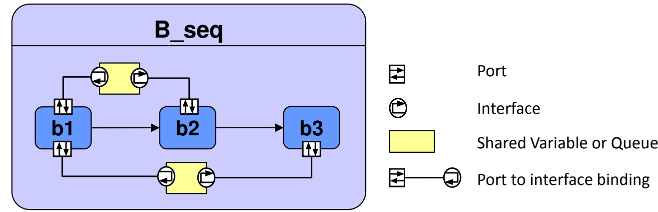


Figure 5.14: Communication between sequentially scheduled (Leaf) Behaviors

$$
\begin{aligned}
& \qquad is_free = \textbf{\textit{true}} \\
& \qquad \textbf{\textit{if}}\ is_waiting == \textbf{\textit{true then notify}}(req) \\
& \qquad \textbf{\textit{wait}}(ack) \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
receive \quad &: \quad \textbf{\textit{void}} \rightarrow Type \\
receive() \quad &= \quad \{ \\
& \qquad \textbf{\textit{if}}\ is_free == \textbf{\textit{false then}} \\
& \qquad\qquad is_waiting = \textbf{\textit{true}} \\
& \qquad\qquad \textbf{\textit{wait}}(req) \\
& \qquad\qquad is_waiting = \textbf{\textit{false}} \\
& \qquad is_full = \textbf{\textit{false}} \\
& \qquad \textbf{\textit{notify}}(ack) \\
& \qquad \textbf{\textit{return}}\ data \\
& \quad \}
\end{aligned}
$$

□

Properties of the Double Handshake Channel, as defined in the SpecC reference implementation [229]:

- in this channel the sender and receiver must meet in a rendezvous to exchange data
- exactly one receiver and one sender thread may use the same channel instance at the same time; if used by more than one sender or receiver, the behavior of the channel is undefined
- the same channel instance may be used multiple times in order to transfer multiple data packets from the sender to the receiver
- using the transceiver interface, the channel may be used bidirectionally
- the sender calls `send()` to send data to the receiver
- the receiver calls `receive()` to receive data from the sender
- the channel operates in rendezvous fashion; a call to `send()` will suspend the sender until the receiver calls `receive()`, and vice versa; when both are ready, data is transferred from the sender to the receiver and both can resume their execution
- calling `send()` or `receive()` may suspend either the sender or the receiver indefinitely

The usage of channels is restricted in the following way:

SEQ and FSM Communication between *sequentially* and *finite-state machine* scheduled (Leaf) Behaviors is restricted to Shared Variables or Queues. Arbitrary read/write access (bidirectional access) is allowed. The usage of Handshake and Double Handshake Channels is not allowed, since its usage will lead to an infinite suspension of either the sender or receiver. Piped Variables are only allowed to be used within a Pipeline. Figure 5.14 provides an example of communication between sequentially scheduled (Leaf) Behaviors.

PIPE Communication between *pipeline* scheduled (Leaf) Behaviors is restricted to Shared Variables, Queues and Piped Variables. The Piped Variable's delay annotations indicates the amount of bridged pipeline stages in the direction of the dataflow. Each delay causes

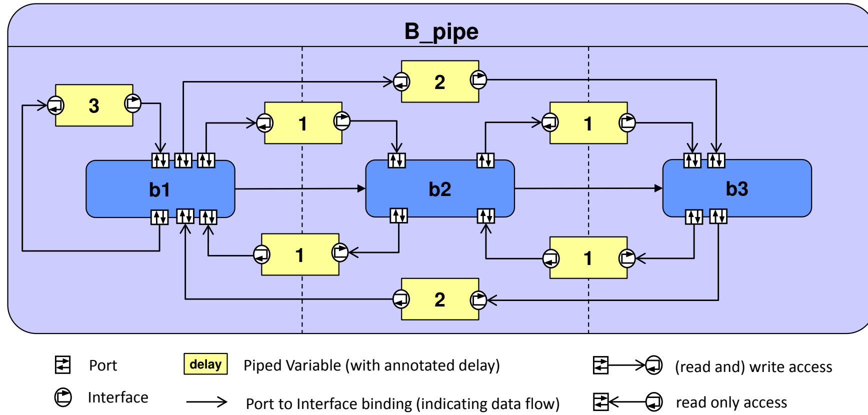


Figure 5.15: Communication between pipeline scheduled (Leaf) Behaviors

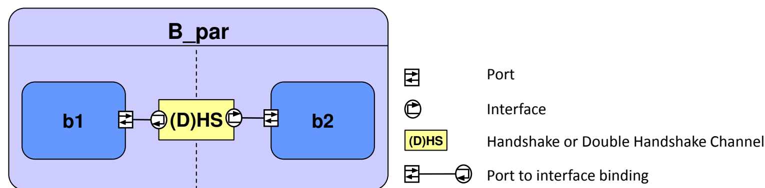


Figure 5.16: Communication between parallel scheduled (Leaf) Behaviors

a FIFO depth of $Delay + 1$ for storing the state of the Variable or Object. The usage of Handshake and Double Handshake Channels is allowed, but special care should be taken in the ramp-up and ramp-down phase of the pipeline. In these phases the usage of Handshake and Double Handshake Communication can lead to an infinite suspension of either the sender or receiver. Figure 5.15 provides an example of communication between pipeline scheduled (Leaf) Behaviors.

PAR Communication between *parallel* scheduled (Leaf) Behaviors can use any kind of channel (except a Piped Variable). To avoid race conditions and possible data inconsistencies, communication shall not be realized by resynchronized Shared Variables or Queues but synchronized Handshake or Double Handshake Channels. Figure 5.16 provides an example of communication between parallel scheduled (Leaf) Behaviors.

5.4.3 Operational Semantics

After presenting the modeling elements of the Behavior Layer in the last section, now for these modeling elements the operational semantics will be defined. In this context the term *Operational Semantics* is used to describe the execution of the language/modeling elements through mapping them to the semantics of timed automata (as introduced in Section 3.3). For the timed automata we use Uppaal notation and graphical representation. For more information about this notation see Section 3.3.2 and Appendix B.

5.4.3.1 Leaf Behavior

The Behavioral Model is an untimed model, i.e. untimed progress of the model can only happen inside Leaf Behaviors. Leaf Behaviors can be of type: regular, state, initial state, end state, and pipeline stage. Inside all these types of Leaf Behaviors the functionality is described using our *Object Model* as described in Section 5.3. The functionality inside Leaf Behaviors is a pure sequential execution model, consisting of a sequence of statements including (data dependent) loops, branches, and calls to routines. This can be formalized as a combination of Call and Control Flow Graphs as introduced in our *Program Graph Representation Model* in Section 3.6.3.

For our tagged signal model each sequential process has a signal $s \in \wp(T \times V)$, where $v \in V$ denotes the state of the leaf behavior, and $t \in T$ is the totally ordered sequence of states, corresponding to the execution of a program.

For each Behavior we define a single entry and a single exit point. That is a Behavior gets activated if the control flow reaches its entry point and is finished when the control flow reaches its exit point. For marking these special entry and exit points in our tagged signal model we introduce the following two special values:

$$\begin{aligned} start &\in V \quad \text{marks the entry point of a Behavior} \\ end &\in V \quad \text{marks the exit point of a Behavior} \end{aligned}$$

with the following associated events $e_{start}, e_{end} \in S$, $t \in T$:

$$\begin{aligned} e_{start} &:= (t, start) \\ e_{end} &:= (t, end) \end{aligned}$$

For each Leaf Behavior B_i we associate a set of signals $S_{B_i} \subseteq S$. Since Leaf Behaviors are not allowed to contain infinite loops, all signals in S_{B_i} contain a totally ordered finite number of events $e \in T \times V$. Let L_{B_i} be the maximum length (number of events) of all signals in S_{B_i}.

Definition 5.4.3.1 (Context):
If the associated control flow graph of the Leaf Behavior contains jumps or loops, different input data or state updates lead to different paths through the control flow. We call these different possible paths contexts and define that the total number of contexts for each Leaf Behavior is finite. This can be proven by inductions since no infinite loops are allowed.

We define the context dependent length of signal $s \subseteq S_{B_i}$ of Leaf Behavior B_i as

$$\begin{aligned} l &: \quad Behavior \times \mathbb{N}_0^+ \rightarrow \mathbb{N}_0^+ \\ l(B_i, context) &= \quad \text{total number of states in } B_i \text{ for this context} \end{aligned}$$

With this definition $L_{B_i} = \max_j\{l(B_i, j)\}$ describes the longest path of executable states in B_i.

We can now define the following context aware function:

Event constructor e: *$Behavior \times \mathbb{N}_0^+ \times \mathbb{N}_0^+ \rightarrow E$ creates an event e for Behavior B, context c and index i. The event consists of tag $t_{B,c,i} \in T$ and value $v_{B,c,i} \in V$: $e(B, c, i) = (t_{B,c,i}, v_{B,c,i})$*

Start constructor $start$: *$Behavior \rightarrow E$ creates an event e_{start} for Behavior B. The event consists of tag $t_B \in T$ and value $v_B := start_B$: $start(B) = (t_B, start_B)$*

End constructor end: *$Behavior \rightarrow E$ creates an event e_{end} for Behavior B. The event consists of tag $t_B \in T$ and value $v_B := end_B$: $end(B) = (t_B, end_B)$*

Tag selector T: *$E \rightarrow T$ returns the tag $t_{B,c,i} \in T$ of event e: $T(e(B, c, i)) = t_{B,c,i}$*

Value selector V: *$E \rightarrow V$ returns the value $v_{B,c,i} \in V$ of event e: $V(e(B, c, i)) = v_{B,c,i}$*

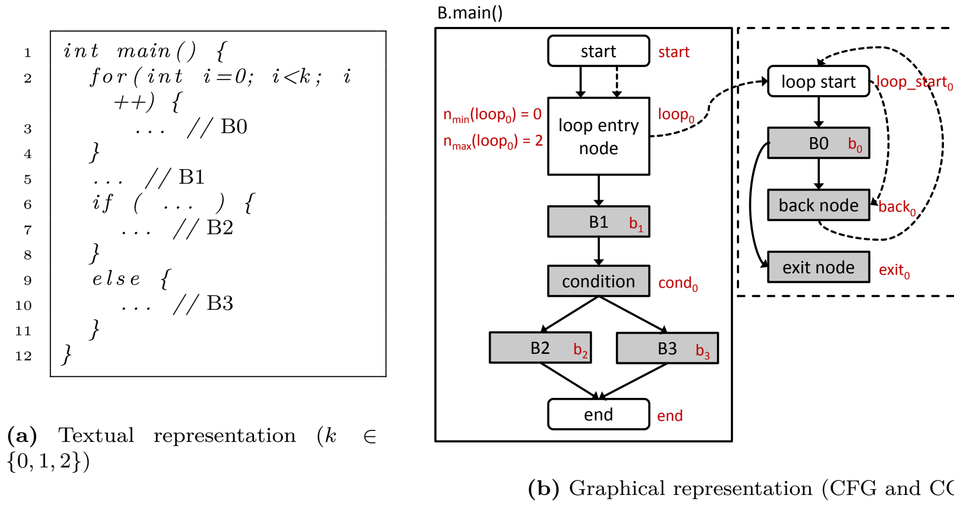
□

The set of signals $S(B_i, j) \subseteq S_{B_i} \subseteq S$ of Leaf Behavior B_i for context j can now be written as:

$$S(B_i, j) := (start(B_i), e(B_i, j, 0), \ldots, e(B_i, j, l(B_i, j) - 1), end(B_i))$$

With the sequential execution relationship we get $s_i < s_{i+1} \Leftrightarrow e_{i,j} < e_{i+1,k}$ for all $0 \leq i < N - 2$, and for all $j, k \in \mathbb{N}_0^+$.

Example (Context aware trace through Leaf Behavior):



(a) Textual representation ($k \in \{0, 1, 2\}$)

(b) Graphical representation (CFG and CG)

Figure 5.17: Example of context generation for Leaf Behaviors without Service Call nodes

Given Behavior B whose programming code is shown in Listing 5.17a its associated CFG and CG, as shown in Figure 5.17b, can be generated. Assuming bounded loop iterations, given by the interval $[n_{min}(loop_0), n_{max}(loop_0)]$ *we can compute all Signals* S_B *of Behavior B by covering all paths of its CFG and CG. For this example we get the following set of signals* $S(B, j) = S_B$, $j \in [0, 5]$*:*

$$
\begin{aligned}
S(B,0) &= (\mathbf{start}, loop_0, b_1, cond_0, b_2, \mathbf{end}) \\
S(B,1) &= (\mathbf{start}, loop_0, b_1, cond_0, b_3, \mathbf{end}) \\
S(B,2) &= (\mathbf{start}, loop_0, loop_start_0, b_0, exit_0, b_1, cond_0, b_2, \mathbf{end}) \\
S(B,3) &= (\mathbf{start}, loop_0, loop_start_0, b_0, exit_0, b_1, cond_0, b_3, \mathbf{end}) \\
S(B,4) &= (\mathbf{start}, loop_0, loop_start_0, b_0, back_0, loop_start_0, b_0, exit_0, b_1, cond_0, b_2, \mathbf{end}) \\
S(B,5) &= (\mathbf{start}, loop_0, loop_start_0, b_0, back_0, loop_start_0, b_0, exit_0, b_1, cond_0, b_3, \mathbf{end}) \\
\\
L(B) &= 10
\end{aligned}
$$

*

As presented in Section 5.4.2 Leaf Behaviors can be hierarchically composed, either in a sequential (SEQ), finite state-machine (FSM), parallel (PAR) or pipelined ($PIPE$) way. Neglecting channel communication and synchronization between (leaf) Behaviors in the first place, we can now define the operational semantics for behavior composition. In the following subsections the execution semantics of allowed behavior compositions is expressed as timed automata. For the sake of simplicity, an example based mapping instead of a formal mapping relation between PSM and timed automata has been chosen.

5.4.3.2 Sequential composition (SEQ)

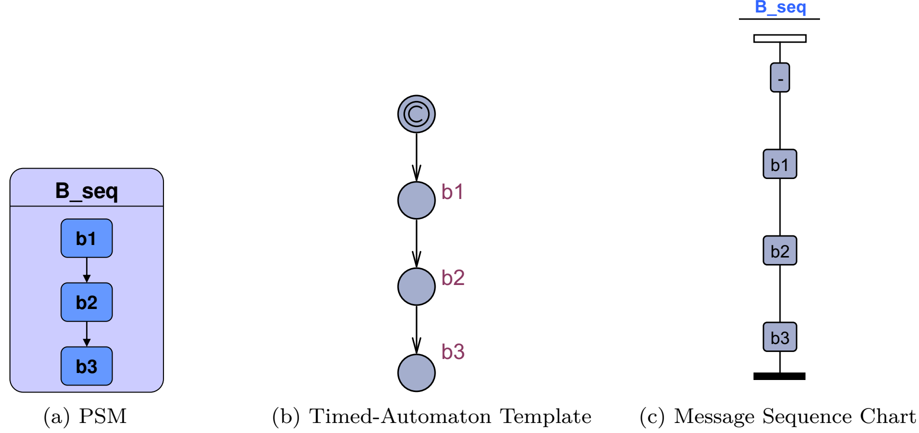
For a sequential composition of Leaf Behaviors $B_0, \ldots, B_N$, written as $B_0 \to B_1 \to \cdots \to B_N$. Applying the sequential process constructor SEQ to the signals $S_0, \ldots, S_N$ we get:

$$
\begin{aligned}
SEQ &: S \times \cdots \times S \to S \\
SEQ(S(B_0, j_0), \ldots, S(B_N, j_N)) &= (S(B_0, j_0), \ldots, S(B_N, j_N))
\end{aligned}
$$

with

$$S(B_i, j) < S(B_{i+1}, k) \quad \forall\, 0 \leq i < N \wedge j, k \in \mathbb{N}_0^+$$

representing a totally ordered composition of signals for all different combinations of contexts.



(a) PSM (b) Timed-Automaton Template (c) Message Sequence Chart

Figure 5.18: Untimed sequential execution

Figure 5.18 shows the mapping of an untimed sequential composition B_{seq} of Behaviors b_1, b_2 and b_3 (5.18a) to a timed automaton template (5.18b) and its associated message sequence chart (5.18c).

5.4.3.3 Finite-state machine composition (FSM)

For a finite-state machine $FSM(S, P, V, F, H, s_0, E)$ composition of Leaf Behaviors $\{B_0, \ldots, B_N\} \in S$, with $s_0 = B_0$ we iteratively apply the FSM signal composition function that is defined for each state S in the following way:

$$\begin{aligned} FSM &: Behavior \times \mathbb{N}_0^+ \rightarrow S \\ FSM(B, i) &= \begin{cases} (S(B, (context, i))) \\ \quad \text{if } B \text{ has no outgoing transitions} \\ (S(B, (context, i)), \bigvee (FSM(B_0, i), \ldots, FSM(B_M, i)) \\ \quad \text{if } B \text{ has outgoing transitions to } B_0, \ldots B_M, \\ \quad i = i + 1 \text{ if target state has incoming transitions} \end{cases} \end{aligned}$$

with the non-deterministic choice function $\bigvee(\cdots)$:

$$\begin{aligned} \bigvee(\cdots) &: S \times \cdots \times S \rightarrow S \\ \bigvee(fsm_0, \ldots, fsm_N) &= fsm_0 \vee \ldots \vee fsm_N \end{aligned}$$

Applying function $FSM(B_0, 0)$ generates a context aware composite signal with all possible execution traces of the finite-state machine. Since a finite-state machine describes a pure sequential execution order of states, the resulting signal is totally ordered, described by:

$$S(B_a, (i, j)) < S(B_b, (k, l)) \text{ if } \exists\, B_a \rightarrow B_b \wedge j \leq l$$

where $i, j, k, l \in \mathbb{N}_0^+$.

Example (Finite State Machine context aware trace):

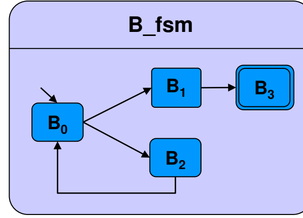
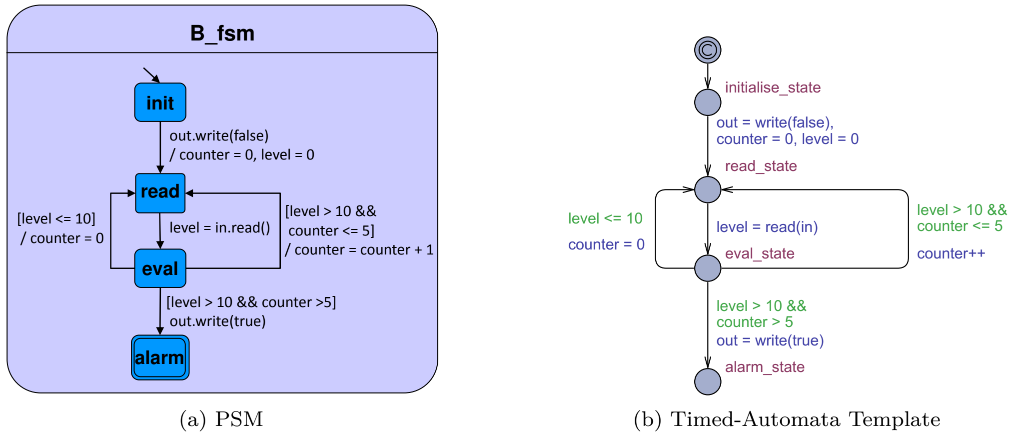


Figure 5.19: Finite State Machine signal trace example

For the state machine, shown in Figure 5.19 we get the following context aware composite signal description:

$$\begin{aligned} FSM(B_0, i) &= (S(B_0, (j_{B_0}, i)), \bigvee(FSM(B_1, i), FSM(B_2, i))) \\ FSM(B_1, i) &= (S(B_1, (j_{B_1}, i)), FSM(B_3, i)) \\ FSM(B_2, i) &= (S(B_2, (j_{B_2}, i)), FSM(B_0, i+1)) \\ FSM(B_3, i) &= (S(B_3, (j_{B_3}, i))) \end{aligned}$$

*



(a) PSM (b) Timed-Automata Template

Figure 5.20: Untimed finite-state machine execution

The mapping to a timed automaton is straightforward and shown in Figure 5.20.

5.4.3.4 Parallel composition (PAR)

For a parallel composition of Leaf Behaviors $B_0, \ldots, B_N$, written as $B_0|B_1|\cdots|B_N$, applying the parallel process constructor PAR to the signals $S_0, \ldots, S_N$ we get:

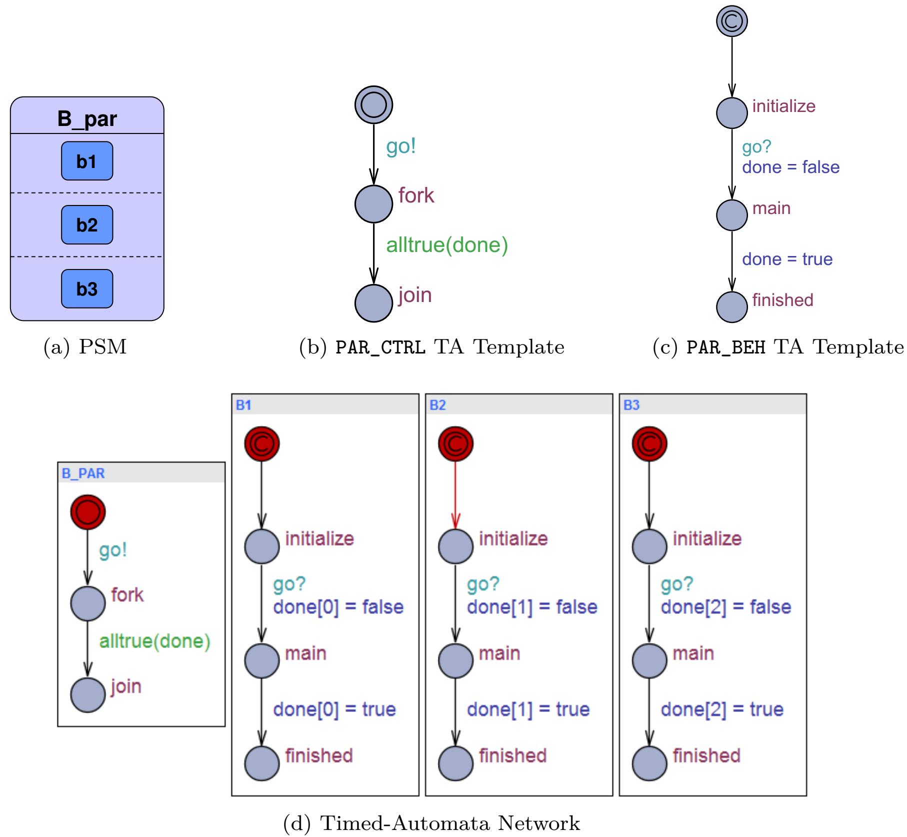
$$\begin{aligned} PAR &: S \times \cdots \times S \to S \\ PAR(S(B_0, j_0), \ldots, S(B_N, j_N)) &= \bigwedge(S(B_0, j_0), \ldots, S(B_N, j_N)) \end{aligned}$$

with the Fork-Join function $\bigwedge(\cdots)$:

$$\begin{aligned} \bigwedge(\cdots) &: S \times \cdots \times S \to S \\ \bigwedge(S_0, \ldots, S_N) &= S_0 \wedge \ldots \wedge S_N \end{aligned}$$

that describes a partial order relationship among the signals $S_0, \ldots, S_N$ in a fork-join semantics:

$$\text{synchronous start (fork)} \quad : \quad T(start(B_0)) = \cdots = T(start(B_N))$$
$$\text{synchronous end (join)} \quad : \quad T(end(B_0)) = \cdots = T(end(B_N))$$



(a) PSM (b) `PAR_CTRL` TA Template (c) `PAR_BEH` TA Template

(d) Timed-Automata Network

Figure 5.21: Untimed parallel execution

Figure 5.21 shows the parallel composition using a fork-join synchronization as timed automata. The TA template

```
PAR_CTRL(urgent broadcast chan &go, bool &done[num_par])
```

implements the fork-join concept (see Figure 5.21b) for a number of `num_par` parallel behaviors. It sends an event to broadcast channel `go` and waits until all shared variables `done[num_par]` are set to `true` using the `alltrue` function

```
bool alltrue(bool a[num_par]) { return forall (i : int[0,num_par-1]) a[i]; }.
```

Each forked Behavior (see Figure 5.21b) is implemented by the TA template

```
PAR_BEH(urgent broadcast chan &go, bool &done).
```

After reception of the `go` event the `main` state is executed. Afterwards `done` is set to `true` for affirming the completion of the `main` function.

Listing 5.1 shows the instantiation of the `PAR_CTRL` and `PAR_BEH` templates in a TA network (see Figure 5.21d) to model the parallel composition of three behaviors as shown in Figure 5.21a.

Figure 5.22 shows the MSC of an arbitrary execution of the TA system from Figure 5.21d. The important property of the fork-join synchronization is illustrated

- by the synchronous start of the `main` states in `B1`, `B2` and `B3` after `B_PAR` fired the `go` event,
- and activation of the `join` state of `B_PAR` after `B1`, `B2` and `B3` have all entered their `finished` state.

```
1  const int num_par ← 3;
2
3  urgent broadcast chan go;
4  bool done[num_par];
5
6  B1 ← PAR_BEH(go, done[0]);
7  B2 ← PAR_BEH(go, done[1]);
8  B3 ← PAR_BEH(go, done[2]);
9  B_PAR ← PAR_CTRL(go, done);
10
11 system B_PAR, B1, B2, B3;
```

Listing 5.1: Timed automata system shown in Figure 5.21d

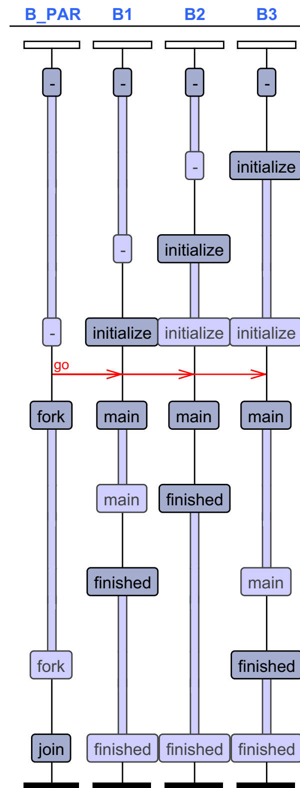


Figure 5.22: Example: Message Sequence Chart of untimed parallel execution (cp. Figure 5.21d)

5.4.3.5 Pipeline composition (PIPE)

A pipeline composition of Leaf Behaviors $B_0, \dots, B_N$, written as $B_0 >> B_1 >> \cdots >> B_N$ is a combination of parallel and sequential behavior composition. In a pipeline each stage is executed in parallel to the other stages. This parallel composition is constrained by a sequential execution order in the activation of the different pipeline stages. In a pipeline composite behavior the number of invocations of each pipeline stage can be bound to I. For a pipeline with N pipeline stages the number of total pipeline iterations or I invocations per stage can be computed by:

$$M \quad : \quad \mathbb{N}_{>0} \to \mathbb{N}_{>0}$$

$$M(I) \quad = \quad \begin{cases} 2N - 1 & \text{if } I \leq N \\ N + I - 1 & \text{if } I > N \end{cases}$$

Applying the pipeline process constructor $PIPE$ to the signals $S_0, \dots, S_N$ for I invocations per stage we get:

$$PIPE : \mathbb{N}_{>0} \times S \times \cdots \times S \to S$$

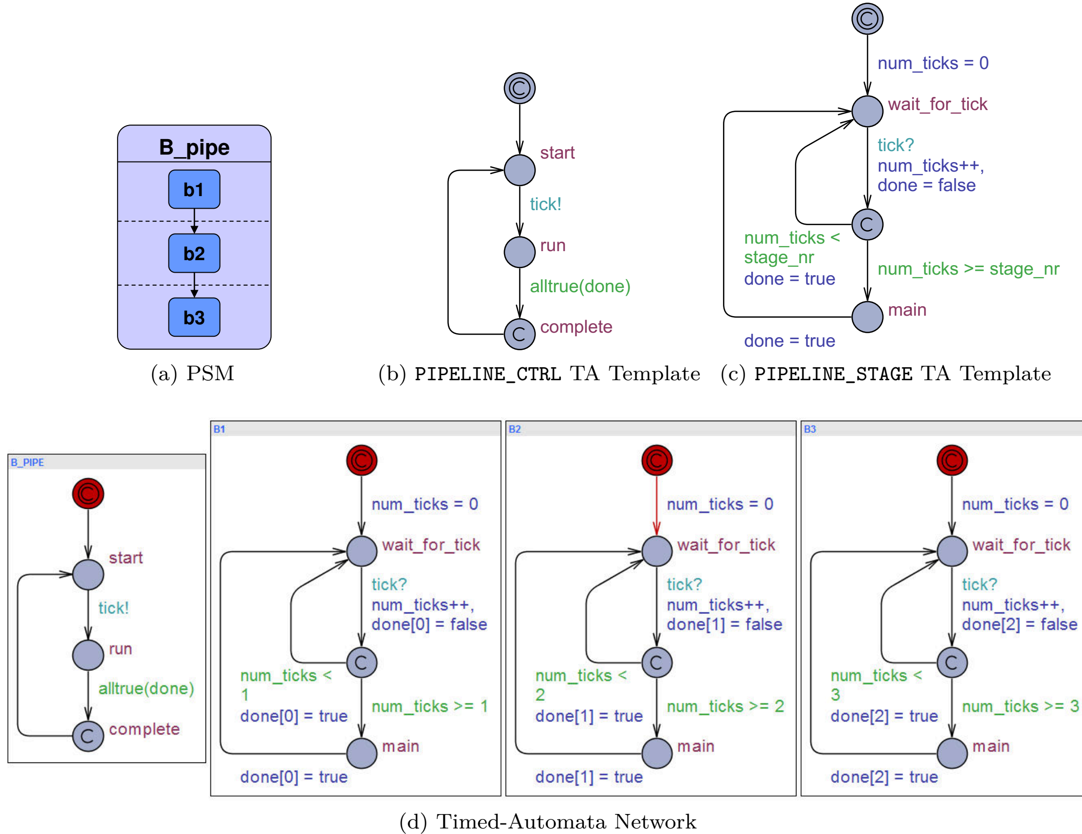
$$
\begin{aligned}
PIPE(I, S(B_0, j_0), \ldots, S(B_N, j_N)) \;=\; & \\
& \Big(\bigwedge(stage(I, S(B_0, j_0), 0), \ldots, stage(I, S(B_N, j_N), 0)), \\
& \bigwedge(stage(I, S(B_0, j_0), 1), \ldots, stage(I, S(B_N, j_N), 1)), \\
& \vdots \\
& \bigwedge(stage(I, S(B_0, j_0), M(I)), \ldots, stage(I, S(B_N, j_N), M(I)))\Big)
\end{aligned}
$$

with the pipeline stage execution function

$$
\begin{aligned}
stage \;&:\; S \times \mathbb{N}_{\geq 0} \rightarrow S \\
stage(I, S(B_i, count[i]), iteration) \;&=\; \begin{cases} S(B_i, count[i]); count[i]++ \\ \quad \text{if } iteration \geq i \wedge count[i] \leq I \\ \bot \\ \quad \text{otherwise} \end{cases}
\end{aligned}
$$

where $i \leq N - 1$, $count[i] \in \underbrace{\mathbb{N}_{\geq 0} \times \cdots \times \mathbb{N}_{\geq 0}}_{i}$.

The mapping of the pipeline composition operational semantics to timed automata will be done in two steps: 1st *unconstrained pipelines* and 2nd *constrained pipelines*.



(a) PSM (b) PIPELINE_CTRL TA Template (c) PIPELINE_STAGE TA Template

(d) Timed-Automata Network

Figure 5.23: Untimed unconstrained pipeline execution

Unconstrained Pipelines Figure 5.23 shows the unconstrained pipeline composition using a fork-join synchronization and pipeline stage as timed automata. The TA template

```
PIPELINE_CTRL(urgent broadcast chan &tick, bool &done[num_stages])
```

implements the fork-join concept (see Figure 5.23b) for a number of `num_stage` pipeline stages. It sends an event to broadcast channel `tick` and waits until all shared variables `done[num_stages]` are set to `true` using the `alltrue` function

```
bool alltrue(bool a[num_stages]) {return forall (i : int[0,num_stages-1]) a[i];}.
```

Each forked Pipeline Stage (see Figure 5.23c) is implemented by the TA template

```
PIPELINE_STAGE(int stage_nr, urgent broadcast chan &tick, bool &done).
```

After reception of the `tick` event the total number of passed ticks is checked against the offset (== `stage_numer`), for modeling the pipeline's ramp-up phase. For the case `num_ticks >= stage_nr` the `main` state is executed. Afterwards `done` is set to `true` for affirming the completion of the `main` function.

```
1  const int num_stages ← 3;
2
3  urgent broadcast chan tick;
4  bool done[num_stages];
5
6  B1 ← PIPELINE_STAGE(1, tick, done[0]);
7  B2 ← PIPELINE_STAGE(2, tick, done[1]);
8  B3 ← PIPELINE_STAGE(3, tick, done[2]);
9  B_PIPE ← PIPELINE_CTRL(tick, done);
10
11 system B_PIPE, B1, B2, B3;
```

Listing 5.2: Timed automata system shown in Figure 5.23d

Listing 5.2 shows the instantiation of the `PIPELINE_CTRL` and `PIPELINE_STAGE` templates in a TA network (see Figure 5.23d) to model the pipeline composition of three behaviors as shown in Figure 5.23a.

Figure 5.25a shows the MSC of an arbitrary execution of the TA system from Figure 5.23d. The two important properties

1. *fork-join synchronization*
   - by the synchronous start of the pipeline stages `B1`, `B2` and `B3` after `B_PIPE` fires the `tick` event,
   - and activation of the `complete` state of `B_PIPE` after `B1`, `B2` and `B3` have all entered their `wait_for_tick` states.
2. *ramp-up phase*
   - after 1st `tick` event `main` state of only `B1` is activated,
   - after 2nd `tick` event `main` states of `B1` and `B2` are activated,
   - and after 3rd `tick` event `main` states of `B1`, `B2` and `B3` are activated

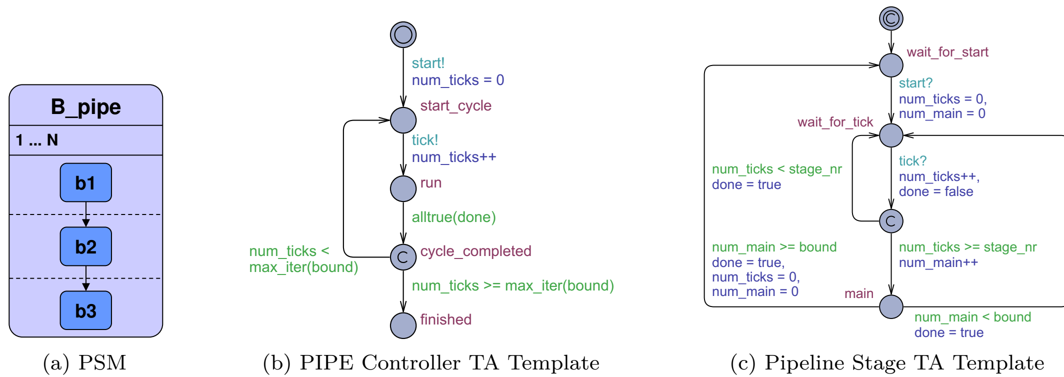
are illustrated.

Constrained Pipelines Figure 5.24 shows the constrained pipeline composition using a fork-join synchronization and pipeline stage as timed automata. The TA template

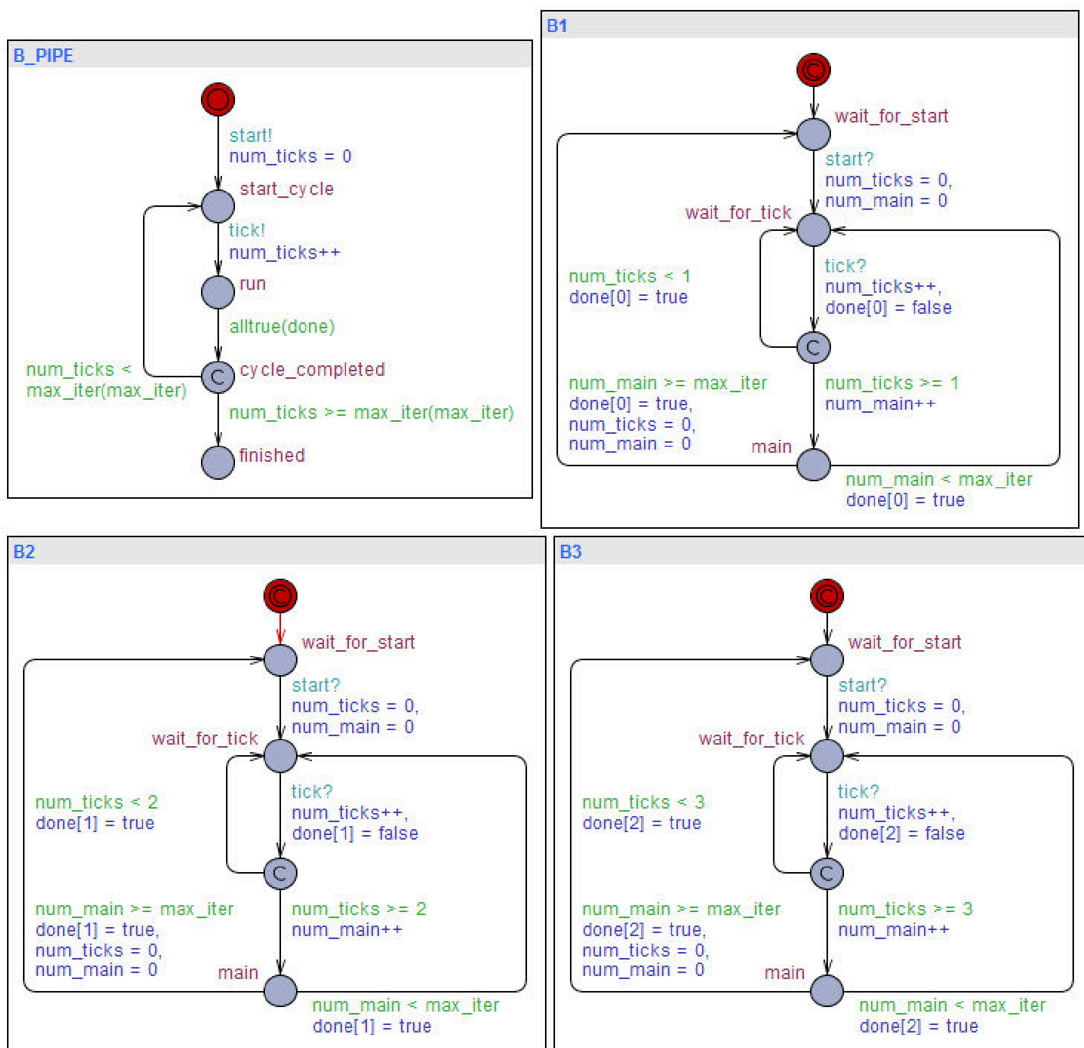
```
PIPELINE_CTRL(int bound, urgent broadcast chan &start,
              urgent broadcast chan &tick, bool &done[num_stages])
```

implements the fork-join concept (see Figure 5.24b) for a number of `num_stage` pipeline stages, each of them being executed `bound` times. Before starting with the execution of any pipeline stage, it sends an event to broadcast channel `start` to initialize all stages. After initialization, it sends an event to broadcast channel `tick` and waits until all shared variables `done[num_stages]` are set to `true` for `max_iter(bound)` times, with

```
int max_iter(int upper_bound){ return num_stages + upper_bound - 1; }.
```



(a) PSM (b) PIPE Controller TA Template (c) Pipeline Stage TA Template



(d) Timed-Automata Network

Figure 5.24: Untimed constrained pipeline execution

Each forked Pipeline Stage (see Figure 5.24c) is implemented by the TA template

```
PIPELINE_STAGE(int stage_nr,
               int bound, urgent broadcast chan &start,
               urgent broadcast chan &tick, bool &done).
```

The `start` event initializes the internal counters `num_ticks` (counting the number of tick event) and `num_main` (counting the number of `main` executions). After reception of the `tick` event the total number of passed ticks is checked against the offset (== `stage_numer`), for modeling the pipeline's ramp-up phase. For the case `num_ticks >= stage_nr` the `main` state is executed. Afterwards `done` is set to `true` for affirming the completion of the `main` function. When the bounded number of `main` executions has been reached, the `wait_for_start` state is entered again.

```
1  const int num_stages ← 3;
2
3  urgent broadcast chan start;
4  urgent broadcast chan tick;
5  bool done[num_stages];
6
7  B1 ← PIPELINE_STAGE(1, max_iter, start, tick, done[0]);
8  B2 ← PIPELINE_STAGE(2, max_iter, start, tick, done[1]);
9  B3 ← PIPELINE_STAGE(3, max_iter, start, tick, done[2]);
10 B_PIPE ← PIPELINE_CTRL(max_iter, start, tick, done);
11
12 system B_PIPE, B1, B2, B3;
```

Listing 5.3: Timed automata system shown in Figure 5.24d

Listing 5.3 shows the instantiation of the `PIPELINE_CTRL` and `PIPELINE_STAGE` templates in a TA network (see Figure 5.24d) to model the constrained pipeline composition of three behaviors as shown in Figure 5.24a.

Figure 5.25b shows the MSC of an arbitrary execution of a constrained pipeline system (see Figure 5.23d) In this example the pipeline constraint, i.e. number of executions per stage, is $I = 1$. The following three important properties

1. *fork-join synchronization*
   - by the synchronous start of the pipeline stages `B1`, `B2` and `B3` after `B_PIPE` fires the `tick` event,
   - and activation of the `complete` state of `B_PIPE` after `B1`, `B2` and `B3` have all entered their `wait_for_tick` states.
2. *ramp-up phase*
   - after 1st `tick` event `main` state of `B1` gets activated for the first time,
   - after 2nd `tick` event `main` state `B2` get activated for the first time,
   - and after 3rd `tick` event `main` state of `B3`, gets activated for the first time.
3. *execution constraint:* `B1`, `B2` and `B3` are executed $I = 1$ times.

are illustrated.

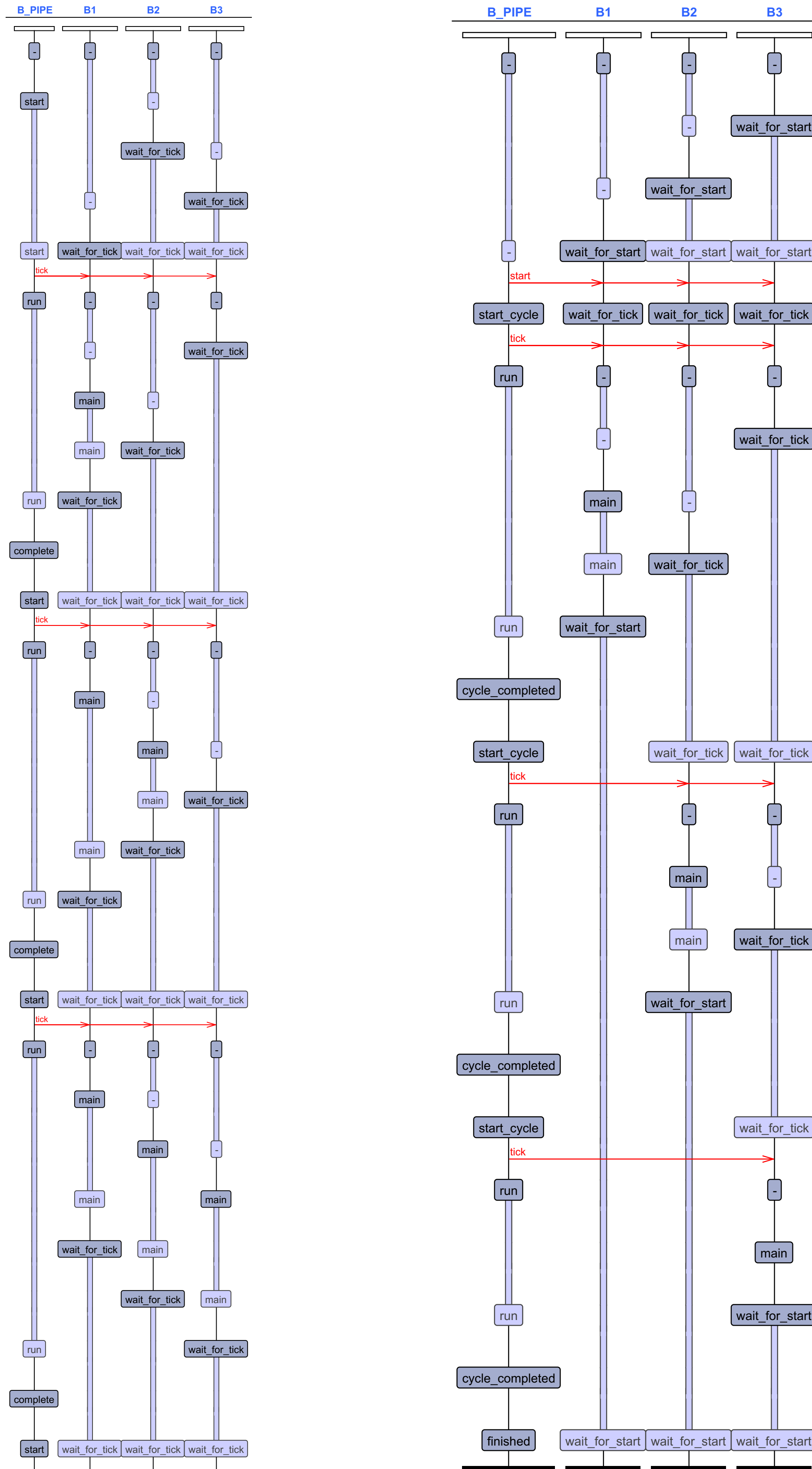
5.4.3.6 Hierarchical composition

The hierarchical composition can be expressed by by successive application of the signal composition functions $SEQ(\cdots)$, $FSM(\cdots)$, $PAR(\cdots)$, and $PIPE(\cdots)$ starting at the root Behavior or Actor, down to the Leaf Behaviors.

To express the hierarchical relationship in timed automata we introduce "stubs" as a generic representation of ancestor and child behaviors. Figure 5.26 shows the `UP_STUB` that represents an ancestor behavior and `DOWN_STUB` that represents a child behavior.

Figure 5.27 illustrates a hierarchical sequential composition. The corresponding MSC example resulting from the TA system shown in Listing 5.4 visualizes the following properties:

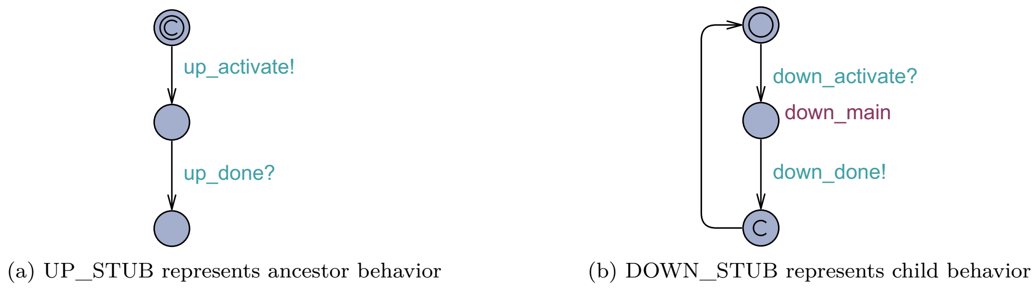
1. `B_seq` starts after receiving the `up_activate` event and terminates, resp. waits for reactivation, after sending the `up_done` event.



(a) unconstrained (∞) pipeline execution (cp. Figure 5.23d)

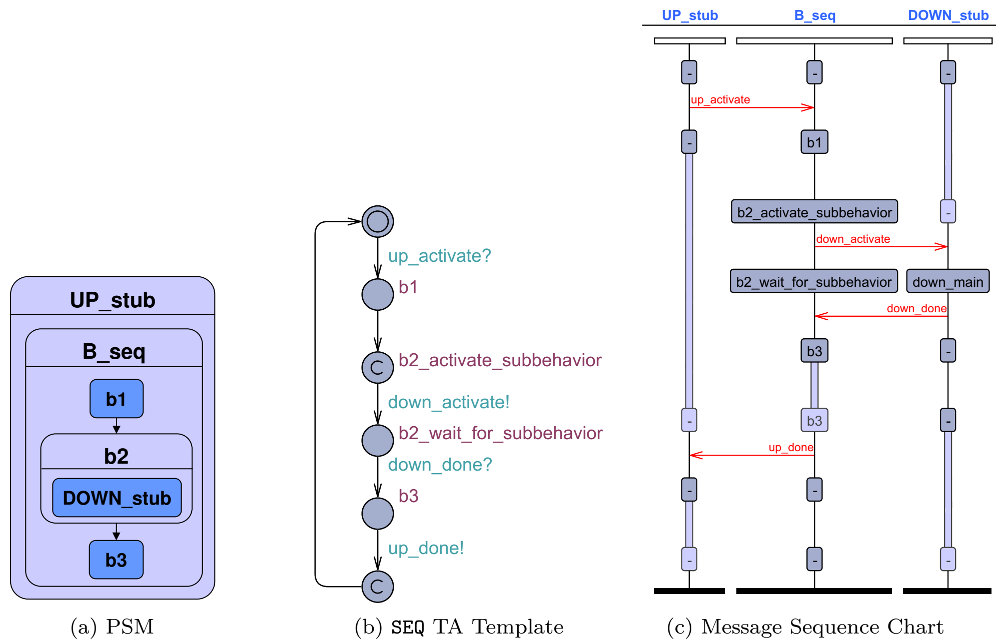
(b) constrained ($I = 1$) pipeline execution (cp. Figure 5.24d)

Figure 5.25: Example: Message Sequence Charts of untimed pipeline executions



(a) UP_STUB represents ancestor behavior (b) DOWN_STUB represents child behavior

Figure 5.26: Stubs used as a generic representation of ancestor and child behaviors



(a) PSM (b) `SEQ` TA Template (c) Message Sequence Chart

Figure 5.27: Hierarchical sequential execution

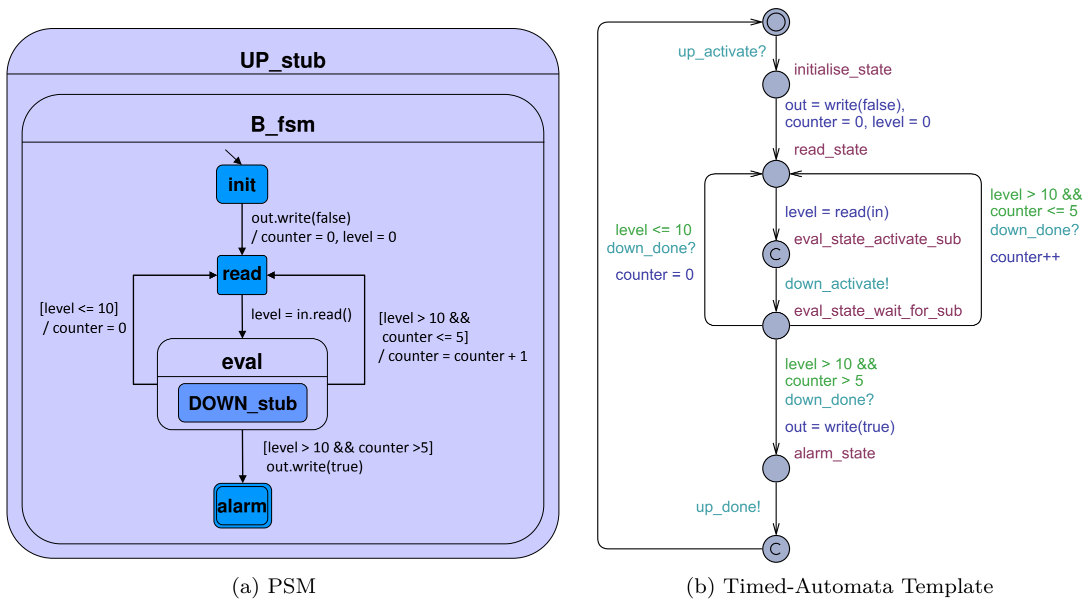
2. The `down_main` behavior of `DOWN_stub` gets activated after completion of `b1` and `b3` gets activated after completion of `down_main`.
3. While `down_main` is active, behavior `B_seq` is waiting in `b2_waiting_for_subbehavior`.

In Figure 5.28 the same style, as used for the hierarchical composition of sequentially scheduled behaviors, has been applied to a finite-state machine:

1. `up_activate?` synchronization before entering the initial state of the FSM
2. `up_done!` synchronization after reaching the final state of the FSM

```
1  urgent chan up_activate;
2  urgent chan up_done;
3  urgent chan down_activate;
4  urgent chan down_done;
5
6  UP_stub   ← UP_STUB(up_activate, up_done);
7  B_seq     ← SEQ(up_activate, up_done, down_activate, down_done);
8  DOWN_stub ← DOWN_STUB(down_activate, down_done);
9
10 system UP_stub, B_seq, DOWN_stub;
```

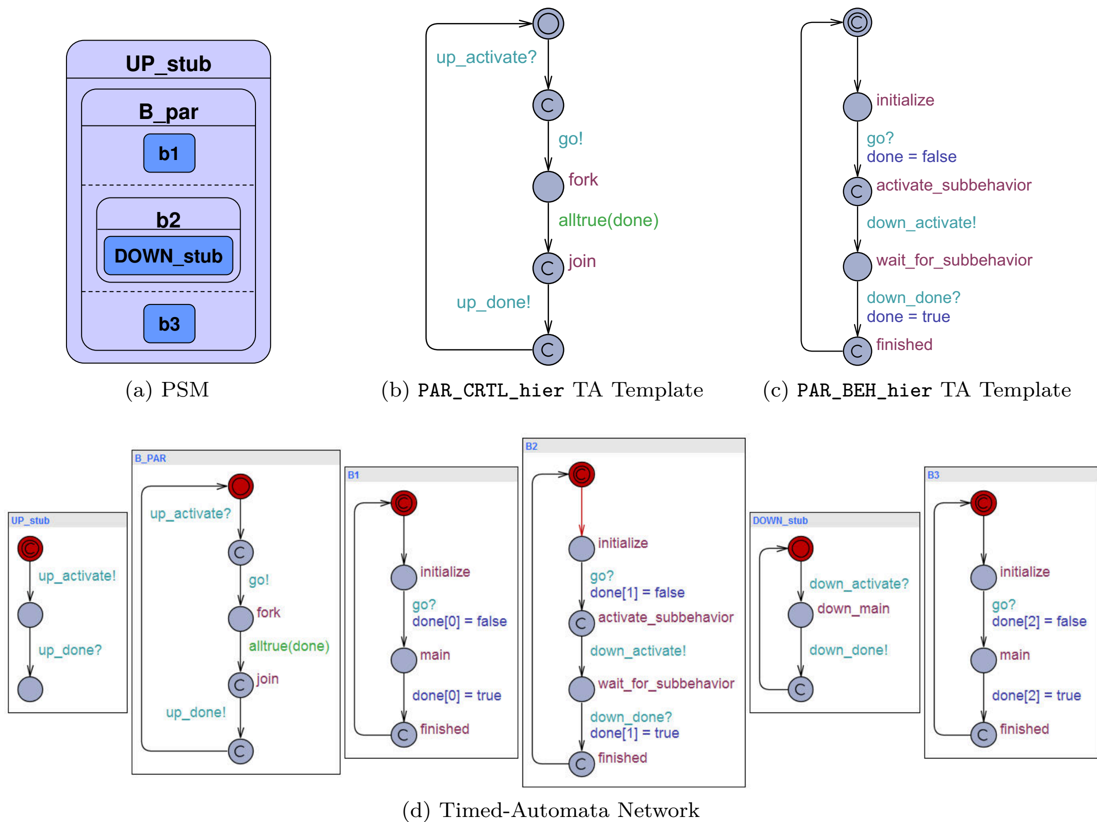
Listing 5.4: Timed automata system for MSC shown in Figure 5.27c



(a) PSM (b) Timed-Automata Template

Figure 5.28: Hierarchical finite-state machine execution

3. inserting committed location with `down_activate!` synchronization into the wait state
4. inserting a wait state with `down_done?` synchronization on all outgoing transitions as replacement for the child behavior



(a) PSM (b) `PAR_CRTL_hier` TA Template (c) `PAR_BEH_hier` TA Template

(d) Timed-Automata Network

Figure 5.29: Untimed hierarchical parallel execution

Figure 5.29 illustrates the hierarchical composition within a parallel composition. In the

`PAR_BEH_hier` behavior it follows the same scheme as for the hierarchical sequential composition. The synchronization with the ancestor is handled through the `up_activate` and `up_done` events in the `PAR_CRTL_hier` behavior.

```
1  urgent chan up_activate;
2  urgent chan up_done;
3  urgent chan down_activate;
4  urgent chan down_done;
5
6  urgent broadcast chan go;
7  bool done[num_par];
8
9  B1 ← PAR_BEH(go, done[0]);
10 B2 ← PAR_BEH_hier(go, done[1], down_activate, down_done);
11 DOWN_stub ← DOWN_STUB(down_activate, down_done);
12 B3 ← PAR_BEH(go, done[2]);
13 UP_stub ← UP_STUB(up_activate, up_done);
14 B_PAR ← PAR_CTRL_hier(go, done, up_activate, up_done);
15
16 system UP_stub, B_PAR, B1, B2, DOWN_stub, B3;
```

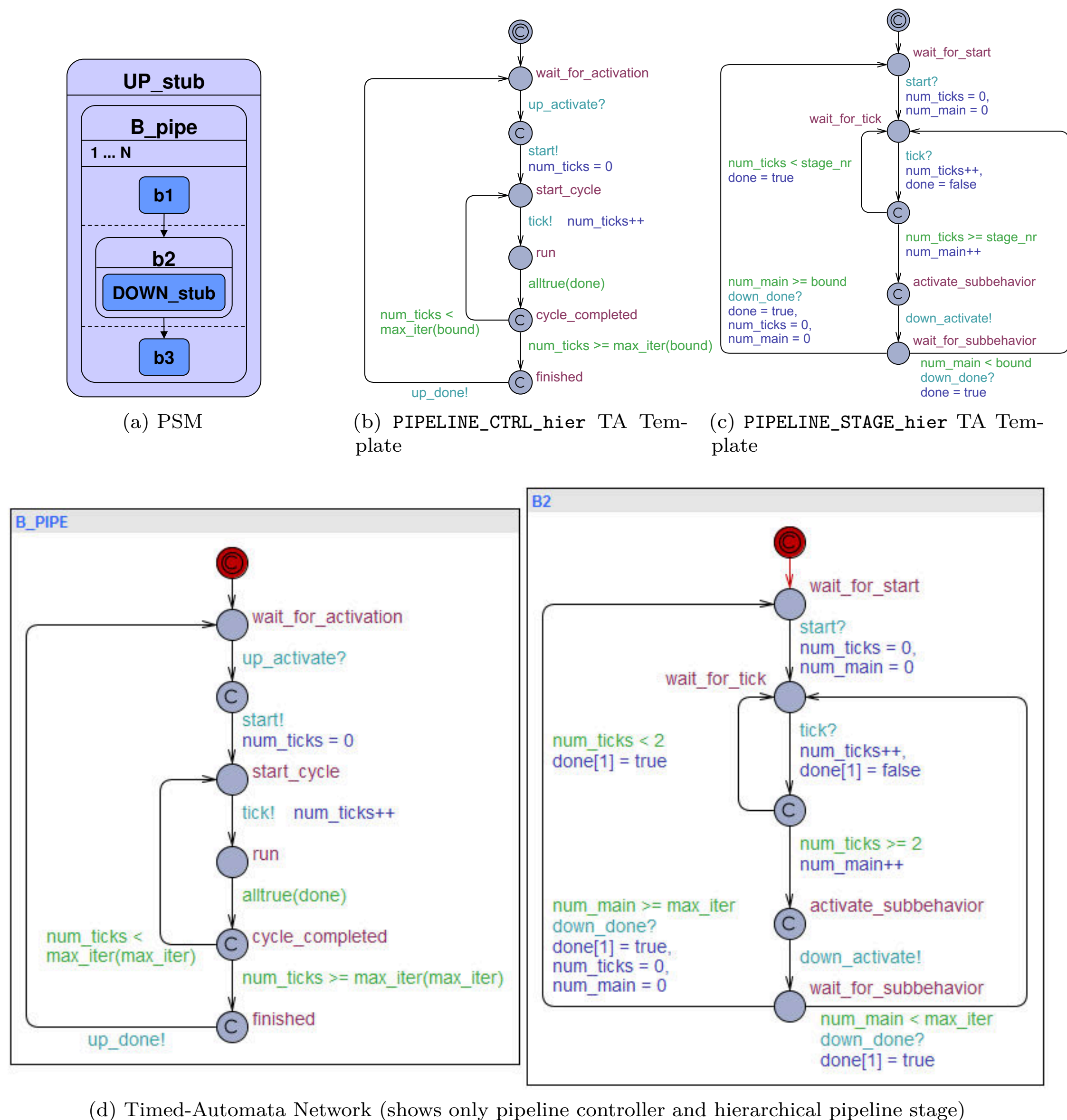
Listing 5.5: Timed automata system shown in Figure 5.29d

Listing 5.5 shows the complete TA network as depicted in Figure 5.29d. Figure 5.31a shows an arbitrary trace example of this TA network. All properties of the parallel composition are retained and `B2` activates `down_main` and finises after execution of `down_main`.

Figure 5.30 depicts the hierarchical composition in a constrained pipeline execution. Listing 5.6 shows the complete TA network. An exemplary trace of this TA network is shown in Figure 5.31b. All properties of the $I = 1$ constrained pipeline execution are retained, while `down_main` is executed while pipeline stage `B2` is active.

```
1  urgent chan up_activate;
2  urgent chan up_done;
3  urgent chan down_activate;
4  urgent chan down_done;
5
6  urgent broadcast chan start;
7  urgent broadcast chan tick;
8  bool done[num_stages];
9
10 B1 ← PIPELINE_STAGE(1, max_iter, start, tick, done[0]);
11 B2 ← PIPELINE_STAGE_hier(2, max_iter, start, tick, done[1], down_activate, down_done);
12 DOWN_stub ← DOWN_STUB(down_activate, down_done);
13 B3 ← PIPELINE_STAGE(3, max_iter, start, tick, done[2]);
14 UP_stub ← UP_STUB(up_activate, up_done);
15 B_PIPE ← PIPELINE_CTRL_hier(max_iter, start, tick, done, up_activate, up_done);
16
17 system UP_stub, B_PIPE, B1, B2, DOWN_stub, B3;
```

Listing 5.6: Timed automata system shown in Figure 5.30d



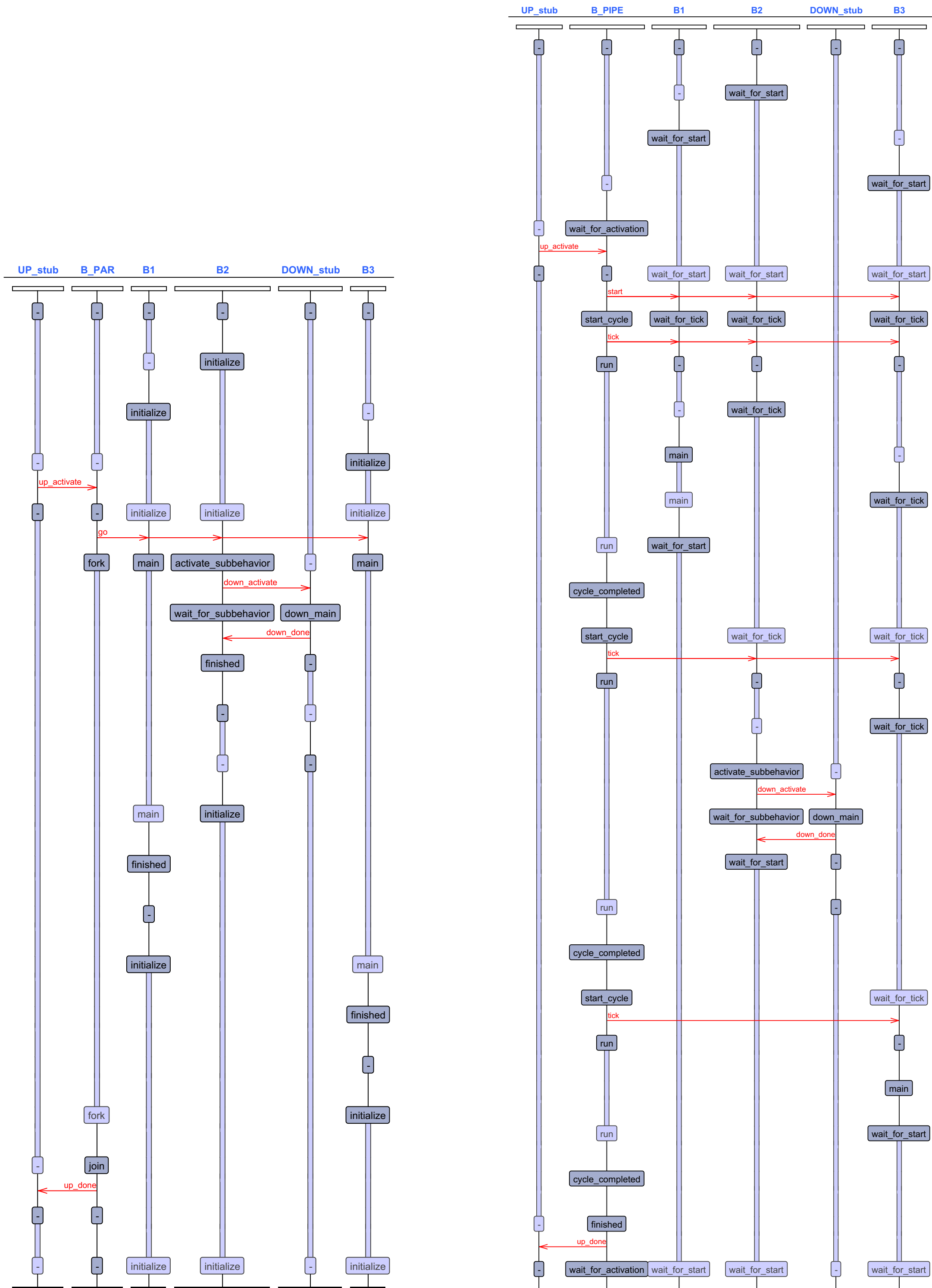
(a) PSM

(b) `PIPELINE_CTRL_hier` TA Template

(c) `PIPELINE_STAGE_hier` TA Template

(d) Timed-Automata Network (shows only pipeline controller and hierarchical pipeline stage)

Figure 5.30: Untimed hierarchical constrained pipeline execution



(a) MSC of untimed parallel execution with hierarchy (cp. Figure 5.29d)

(b) MSC of untimed constrained pipeline executin ($I = 1$) with hierarchy (cp. Figure 5.30d)

Figure 5.31: Example: Message Sequence Charts of untimed parallel and pipelined executions with hierarchy

5.4.3.7 Communication

For expressing communication among leaf behaviors, we extend the definition of Call Graphs (see Definition 3.6.3.4) by adding a new node type for Service Calls via Ports on Channels' interface functions.

Definition 5.4.3.2 (Extended Call Graph):
An extended call graph connects call nodes and and service call nodes with start nodes. Calls, Services, and Starts constitutes the nodes and target restricted to Calls defines the intra-procedural edges, while target restricted to Services defines the port to interface calls on Channels. The linkage between start and

- *call nodes is established by adding edges from start to call nodes for each routine.*
- *service nodes is established by adding edges from start to service nodes for each routine.*

Formally, an extended call graph is defined as: $CG = (\hat{V}, \hat{E})$*, with*

1. *the nodes* $\hat{V} = Calls \cup Services \cup Start$*,*
2. *and the edges* $\hat{E} \subseteq \hat{V} \times \hat{V}$*, where* $\hat{E}$ *is defined as:*

$$\begin{array}{rcl} \hat{E} & := & \{(c,s)\colon c \in Calls, s \in target(c)\} \cup \\ & \bigcup_{f \in F} & \{(s,c)\colon s \in Starts, c \in Calls \cup Services\colon \exists s \rightarrow^{*}_{CFG_f} c\} \end{array}$$

□

We also need to extend the definition of artificial empty nodes from Definition 3.6.3.9.

Definition 5.4.3.3 (Extended Artificial Empty Nodes):
For providing special hooks for analysis and timing annotation of service calls to Channels we introduce in addition to Definition 3.6.3.9 the following additional artificial empty nodes as special locations in the CFG and CG. At each service call invocation two additional nodes are inserted:

- *service call node: linked with start node in the CG. Represents the call to a service via a port to a Channel. Calls to Channels are blocking calls.*
- *service return node: indicates the return of a called service of a Channel*

Services are augmented with the following empty nodes:

- *start node: Service begin with a start node*
- *exit node: returning control flow to the caller is gathered in a unique exit node.*

For each call to a Channel we define a single entry and a single exit point. I.e. a service call on a Channel gets requested when the control flow of a Behavior reaches a Service Call Node. After the requested service call has been performed it returns to the control flow of the Behavior through the Service Return Node. For marking these special call and return nodes in our tagged signal model we introduce the following special values:

$$\begin{array}{rcll} call & \in & V & \textit{marks the entry point of a Service Call (Request)} \\ return & \in & V & \textit{marks the return point of a Service Call} \end{array}$$

with the following associated events: $e_{call}, e_{return} \in E, t \in T$*:*

$$\begin{array}{rcl} e_{call} & := & (t, call) \\ e_{return} & := & (t, return) \end{array}$$

Since Service Calls are context dependent, which means that their execution might be data or state dependent. For this purpose we define the following context aware index functions:

$$\begin{array}{rcl} call & : & Behavior \times \mathbb{N}_0^+ \times Channel \times \mathbb{N}_0^+ \rightarrow E \\ call(B, context, Ch, nr) & = & (t_{B,context,Ch,nr}, call_{B,context,Ch,nr}) \end{array}$$

$$\begin{array}{rcl} return & : & Behavior \times \mathbb{N}_0^+ \times Channel \times \mathbb{N}_0^+ \rightarrow E \end{array}$$

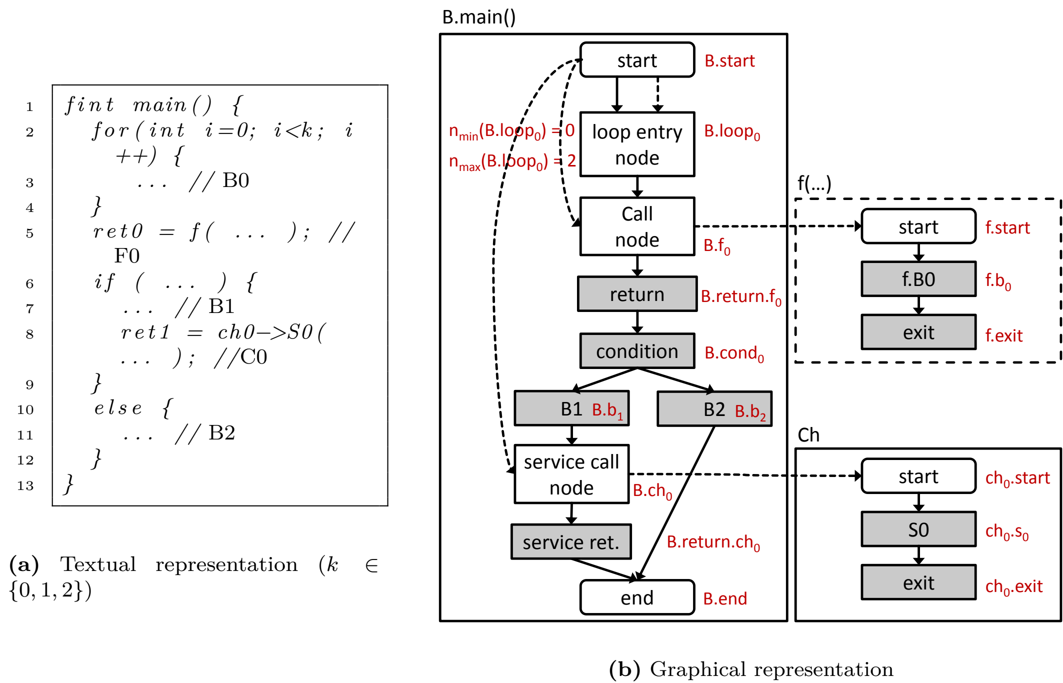
$$return(B, context, Ch, nr) = (t_{B,context,Ch,nr}, return_{B,context,Ch,nr})$$

These index functions associate Service Call and Service Return nodes of a Behavior B*, with a context (in which they appear), the Channel* Ch *the service call is requested from, and the consecutive number of service requests (starting with 0) on* Ch *in this context.* □

Service Calls on Channels have the following basic properties:

1. Service Calls always return (unless there is a deadlock, due to inconsistent specification, which is considered as a design error)
2. Service Calls are blocking. I.e. the control flow of the calling Behavior's context is not allowed to pass any other node until communication is finished

Example (Context aware trace of Leaf Behavior with Channel communication):



(a) Textual representation ($k \in \{0, 1, 2\}$)

(b) Graphical representation

Figure 5.32: Example of context generation for Leaf Behaviors with Service Call nodes

Given Behavior B *whose programming code is shown in Listing 5.32a its associated CFG and CG, as shown in Figure 5.32b, can be generated. Assuming bounded loop iterations, given by the interval* $[n_{min}(loop_0), n_{max}(loop_0)]$ *we can compute all Signals* S_B *of Behavior* B *by covering all paths its CFG and CG. For this example we get the following set of signals* $S(B, j) = S_B$, $j \in [0, 5]$*:*

$$\begin{aligned}
S(B,0) = (&B.start,\\
&B.loop_0,\\
&B.f_0, f.start, f.B_0, f.exit, B.return.f_0,\\
&B.cond_0,\\
&B.b_1,\\
&B.ch_0, ch_0.start, ch_0.s_0, ch_0.exit, B.return.c_0,\\
&B.end)\\
S(B,1) = (&B.start,\\
&B.loop_0,\\
&B.f_0, f.start, f.B_0, f.exit, B.return.f_0,
\end{aligned}$$

$$
\begin{aligned}
& B.cond_0, \\
& B.b_2, \\
& B.end) \\
S(B,2) = (& B.start, \\
& B.loop_0, B.loop_start_0, B.b_0, B.exit_0, \\
& B.f_0, f.start, f.B_0, f.exit, B.return.f_0, \\
& B.cond_0, \\
& B.b_1, \\
& B.ch_0, ch_0.start, ch_0.s_0, ch_0.exit, B.return.c_0, \\
& B.end) \\
S(B,3) = (& B.start, \\
& B.loop_0, B.loop_start_0, B.b_0, B.exit_0, \\
& B.f_0, f.start, f.B_0, f.exit, B.return.f_0, \\
& B.cond_0, \\
& B.b_2, \\
& B.end) \\
S(B,4) = (& B.start, \\
& B.loop_0, B.loop_start_0, B.b_0, B.back_0, B.loop_start_0, B.b_0, B.exit_0, \\
& B.f_0, f.start, f.B_0, f.exit, B.return.f_0, \\
& B.cond_0, \\
& B.b_1, \\
& B.ch_0, ch_0.start, ch_0.s_0, ch_0.exit, B.return.f_0, \\
& B.end) \\
S(B,5) = (& B.start, \\
& B.loop_0, B.loop_start_0, B.b_0, B.back_0, B.loop_start_0, B.b_0, B.exit_0, \\
& B.f_0, f.start, f.B_0, f.exit, B.return.f_0, \\
& B.cond_0, \\
& B.b_2, \\
& B.end)
\end{aligned}
$$

*

In the following paragraphs communication in the Behavior Layer model is mapped to the semantics of Timed Automata.

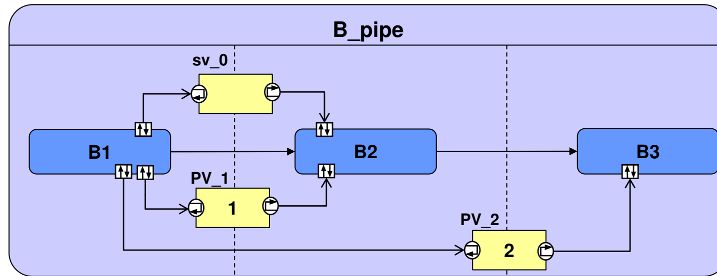
Shared and Piped Variable A Shared Variable is mapped to a global variable in timed automata, see Listing 5.8 (declaration in line 8 and TA template accesses by reference in line 17 and 18).

Piped Variables require their own automata as shown in Figure 5.33b and its corresponding Listing 5.8. Three pipeline stages `B1-B3` are connected through shared and piped variables as depicted in Figure 5.33a. `PV_1` with depth 1 connects `B1` and `B2` and crossed one pipeline stage boundary, while `PV_2` with depth 2 connects `B1` and `B3` and crosses two pipeline boundaries.

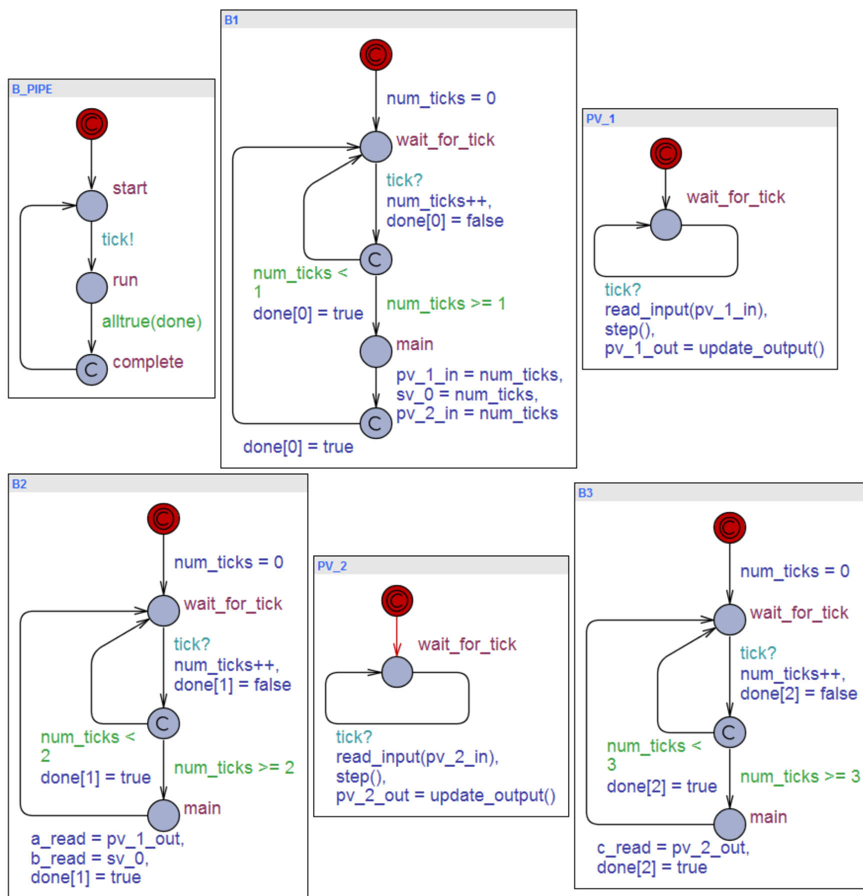
```
1 Type data[depth+1];
2
3 void step() { for (i : int [0,depth−1]) { data[depth−i] ← data[depth−(i+1)]; } }
4
5 void read_input(Type in) { data[0] ← in; }
6 int update_output() { return data[depth]; }
```

Listing 5.7: Piped Variable TA template local functions

Each piped variable TA becomes activated through the global pipeline controller's `tick` broadcast event. Upon synchronization new input data is stored at the beginning of the internal array `read_input`, the internal array is shifted by one `step` and the output gets updated by



(a) PSM



(b) TA system

Figure 5.33: Shared and Piped Variable TA Example

# tick	sv_0	pv_1_out	pv_2_out
1	1	0	0
2	2	1	0
3	3	2	1
4	4	3	2
⋮	⋮	⋮	⋮

Table 5.1: Shared Variable and Piped Variable outputs after number of `ticks`

picking out the last element of the internal array `update_output`. All Piped Variable TA template's local functions are shown in Listing 5.7.

```
1  typedef int Type;
2  const int num_stages ← 3;
3
4  urgent broadcast chan tick;
5  bool done[num_stages];
6
7  // shared variable
8  int sv_0;
9
10 // inputs and outputs to piped variable PV_1
11 int pv_1_in;
12 int pv_1_out;
13 // inputs and outputs to piped variable PV_2
14 int pv_2_in;
15 int pv_2_out;
16
17 B1 ← PIPELINE_STAGE_B1(1, tick, done[0], pv_1_in, sv_0, pv_2_in);
18 B2 ← PIPELINE_STAGE_B2(2, tick, done[1], pv_1_out, sv_0);
19 B3 ← PIPELINE_STAGE_B3(3, tick, done[2], pv_2_out);
20 B_PIPE ← PIPELINE_CTRL(tick, done);
21 PV_1 ← PIPED_VARIABLE(1, tick, done, pv_1_in, pv_1_out);
22 PV_2 ← PIPED_VARIABLE(2, tick, done, pv_2_in, pv_2_out);
23
24 system B_PIPE, B1, PV_1, B2, PV_2, B3;
```

Listing 5.8: Timed automata system shown in Figure 5.33b

Table 5.1 illustrates the assignment of the Shared Variable and Piped Variables at the end of each `tick`. Shared Variables immediately take the newly assigned value, `PV_1` delays its input value by one tick and `PV_2` delays its output value by two ticks.

Queue The Queue is similar to a shared variable: it's a shared array with additional bookkeeping to implement a ring-buffer on the linear array. Listing 5.9 shows the shared data structure `Queue` (line 3-8) and the bookkeeping functions `put`, `get`, `is_full` and `is_empty` as defined in Definition 5.4.2.7.

Figure 5.34 shows the example of using a Queue between parallel behaviors `B1` and `B2`. Listing 5.10 shows the corresponding system instantiation with Queue `Q` (line 7) of size 4 (line 2).

After activation, `B1` attempts to write the integer sequence $\{0, \dots, 10\}$ into Queue `Q`. Before writing into the Queue, the filling level is checked using the `is_full` function. If the Queue is full, the `put` call is deferred until the receiver has consumed at least one element from the Queue (`!is_full` returns true). `B2` acts as the receiver and reads from the Queue `Q` using the `get` function. The receiver only reads from the Queue when there is at least on element available (`!is_empty` returns true).

Handshaking The semantics of Handshake and Double Handshake Channels can be expressed by Uppaal timed automata sender and receiver synchronization. Table 5.2 summarizes the

```
typedef int Type;

typedef struct {
  int put_index;
  int get_index;
  int num_elements;
  Type data[QUEUE_SIZE];
} Queue;

void put(Queue &q, Type data) {
  q.data[q.put_index] ← data;
  if (q.put_index == (QUEUE_SIZE − 1)) { q.put_index ← 0; }
  else { q.put_index ← q.put_index + 1; }
  q.num_elements ← q.num_elements + 1;
}

Type get(Queue &q) {
  Type tmp ← q.data[q.get_index];
  if (q.get_index == (QUEUE_SIZE − 1)) { q.get_index ← 0; }
  else { q.get_index ← q.get_index + 1; }
  q.num_elements ← q.num_elements − 1;
  return tmp;
}

bool is_full (Queue q) { return q.num_elements == QUEUE_SIZE; }
bool is_empty(Queue q) { return q.num_elements == 0; }
```

Listing 5.9: Queue definition

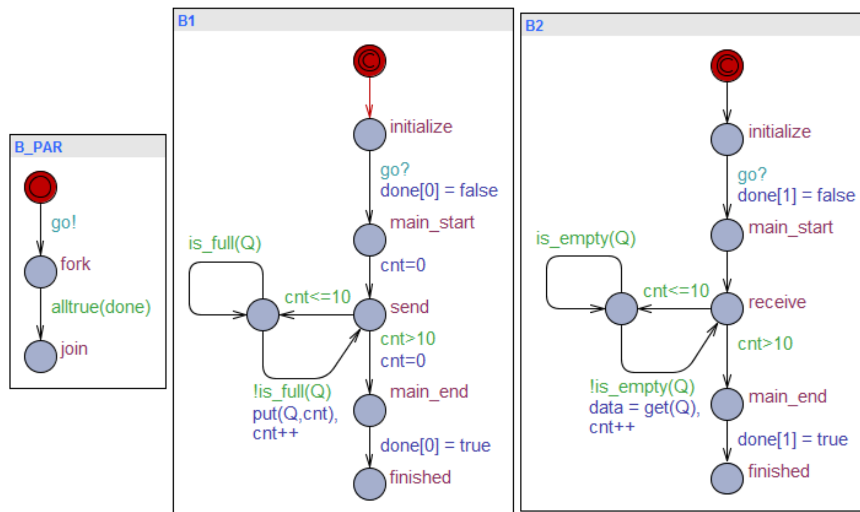


Figure 5.34: TA Queue system example

mapping of Handshake and Double Handshake Channels to Timed Automata sender and receiver synchronization as introduced in Section 3.3.3.

5.5 Application Layer

5.5.1 Introduction

The *Application Layer* focuses on computational structuring of the *Behavior Layer* model. It is an intermediate model introducing computational containers called *Actors* and *Shared Objects* that model communication through method interface calls. The system is modeled as a set of parallel, communicating actors. A shared resource is called *Shared Object*, which equips a user-defined class with specific synchronization facilities and provides a set of services. *Shared*

```
1  const int num_par ← 2;
2  const int QUEUE_SIZE ← 4;
3
4  urgent broadcast chan go;
5  bool done[num_par];
6
7  Queue Q;
8  B1 ← PAR_B1(go, done[0], Q);
9  B2 ← PAR_B2(go, done[1], Q);
10 B_PAR ← PAR_CTRL(go, done);
11
12 system B_PAR, B1, B2;
```

Listing 5.10: TA Queue system as shown in Figure 5.34

Channel	Interface(s)	Timed Automata
Handshake	`send_if`, `receive_if`	single one-way
Handshake	`send_receive_if`	two one-way
Double Handshake	`send_if`, `receive_if`	asymmetric two-way
Double Handshake	`send_receive_if`	symmetric two-way

Table 5.2: Mapping Handshake and Double Handshake Channels to Timed Automata sender and receiver synchronization

Objects are inspired by the *Protected Objects* known from Ada [183, 60] and automatically guarantee mutual exclusive access from different actors.

Synchronization is performed by arbitrating concurrent accesses and a special feature called *Guarded Methods*, that can be used to block the execution of a method until an optional, user-defined condition evaluates to true.

The main properties of the Application Layer can be summarized as:

- Main focus on structuring functionality (functional behavior) into schedulable containers: Actors and Shared Objects
- Modeling style of Actors and Shared Objects ranges from HW/SW implementation independent over HW/SW retargetable to HW/SW dependent implementation.
- Supports the separation of communication and behavior through Port-Interface-Binding and method-based communication between Actors and Shared Objects
- Supports timing (back-)annotations of Actors and Shared Objects
- The model is executable
- Supports timing assertions to monitor (data-dependent) timing behavior

In the following refinement step computational containers (Actors and Shared Objects) of the Application Layer are mapped to processing elements of the *Virtual Target Architecture Layer* enabling synthesis of the application on the targeted hardware platform.

In the following all modeling primitives on the Application Layer are introduced. This following denotational model only covers the required aspects for the description of the mapping to the Virtual Target Architecture in Section 5.6 and abstracts from the details implicitly contained in the C++-based implementation model.

5.5.2 Modeling Elements

The Application Layer is an executable parallel object-oriented model and consists of the following modeling elements:

- **Module** is a pure structural element (does not model any behavior/functionality). Modules can be hierarchically composed (i.e. a Module my contain other Modules). Modules can

contain Actors and Shared Objects (see below). Modules have Ports (see Definition 5.4.2.1) to forward communication from the Ports of Actors and Sub-Modules to the outside. Each module is a unique entity object. Modules cannot be copied or passed by value.

- **Actor** is a specialized Module and container for functionality. An Actor may contain a single thread of execution or multiple threads of execution. Actors contain Behaviors (see Definition 5.4.2.2) from the Behavioral Layer to express functionality. Actors cannot be hierarchically composed (i.e. and Actor is not allowed to contain any other Actors). An Actor has Ports (see Definition 5.4.2.1) to forward Ports of the inner Behavior(s) to the outside. Each Actor is a unique entity object. Actors cannot be copied or passed by value.
- **Passive Object** (the instance of a Class, see Definition 5.3.4.2) basically encapsulates data and operations on it. Passive Objects can be cloned, copied and passed by value. Objects can be derived from other Objects as described in Section 5.3.
- **Shared Object** is a container for a passive Object. Selected methods (called services) of the inner passive object can be exported through interfaces. Services can be associated with a side-effect free Boolean condition (called Guard) on the passive Object's internal state. A service is only available if its Guard evaluates to true. Shared Object services can be called from within the Behavior(s) of Actors through Ports. When multiple Actors or parallel scheduled Behaviors inside one Actor are requesting services of a Shared Object a dynamic and customizable access arbitration is performed (i.e. Shared Object service accesses are always mutual exclusive). Each Shared Object is a unique entity object. Shared Objects cannot be copied or passed by value. From within a service or any other method of the Shared Object's internal passive Object, no other Shared Object services can be called.

In the following subsection these modeling elements will be defined and described in more detail. The Operational Semantics of Actors and Shared Objects is described in Section 5.5.6.

5.5.2.1 Actor

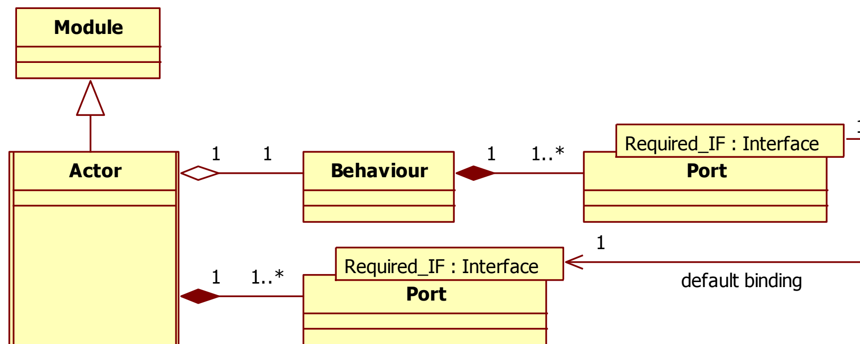


Figure 5.35: Meta-Model of Actors

Definition 5.5.2.1 (Actor):
Figure 5.35 gives an overview of the Actor Meta-Model. An Actor is a tuple

$$Actor = \left[Behavior, \overline{Port}\right], \text{ where}$$

1. *Behavior* $\in$ **Behavior** *is the Behavior of this Actor. Depending on the Behaviors properties we can classify an actor in the following way:*

   Active Sequential *restricts the Behavior and all of its sub Behaviors to:*

   (a) $Type = \{regular\}$
   (b) $Composite = \{SEQ\}$

An Active Sequential Actor only contains a hierarchical sequential compositions, executed in an infinite loop. When the execution of the last sequential behavior of the the root behavior has been finished, the execution of the first sequential behavior of the root behavior is started again. The number of parallel sets for an active sequential actor A is $|par_set(A.Behavior)| = 1$.

Active Parallel *restricts the Behavior and all of its sub Behaviors:*

(a) $Type = \{regular, pipeline_stage\}$
(b) $Composite = \{SEQ, PAR, PIPE\}$

An Active Parallel Actor contains an arbitrary hierarchical mixture of sequential, parallel, and pipeline compositions, executed in an infinite loop. When the execution of the the root behavior has been finished it is started again. The degree of parallelism inside an Active Parallel Actor A is $|par_set(A.Behavior)| \geq 1$.

Reactive Sequential *restricts the Behavior and all of its sub Behaviors to:*

(a) $Type = \{regular, state, initial_state, end_state\}$
(b) $Composite = \{SEQ, FSM\}$

A Reactive Sequential Actor only contains a hierarchical mixture of sequential and finite-state machine compositions, executed in an infinite loop. When the execution of the the root behavior has been finished it is started again. The number of parallel sets for a reactive sequential actor A is $|par_set(A.Behavior)| = 1$. *The term "reactive" describes the ability to use the reactive behavior of the FSM behavior. Transitions in FSM behaviors are taken when the guard condition is fulfilled and the blocking read operation on an in or inout port returns.*

Reactive Parallel *relieves all restrictions on the Behavior and all of its sub Behaviors:*

(a) $Type = \{regular, state, initial_state, end_state, pipeline_stage\}$
(b) $Composite = \{SEQ, FSM, PAR, PIPE\}$

A Reactive Parallel Actor has no restrictions on its behaviors, but also executed in an infinite loop. For a reactive parallel actor A is $|par_set(A.Behavior)| \geq 1$.

2. $\overline{Port} = \{p_0, \cdots, p_N\}$ *is a copy of the set of ports as defined by the Behavior of Actor A, with* $A.\overline{Port} \equiv A.Behavior.\overline{Port}$. *There is a default binding relation between these ports:*

$$\forall 0 \leq i < N : bp_i \rhd mp_i$$

with $bp_i \in A.Behavior.\overline{Port}$ *and* $mp_i \in A.\overline{Port} \quad \forall 0 \leq i < N$.

□

5.5.2.2 Application Layer System

Definition 5.5.2.2 (Application Layer System):
The Application Layer System (ALS) defines the boundary of the system to be designed. Figure 5.36 gives a graphical overview of the Application Layer System Meta-Model. An Application Layer System is a tuple $ALS = \left[\overline{Actor}, \overline{Channel}, \overline{Port}, \overline{Binding}\right]$ *with*

1. $\overline{Actor} = \{a_0, \cdots, a_N\}$ *is a set of Actors* $a_i \in \textbf{Actor}$ *with* $0 \leq i \leq N$ *and* $N \geq 0$.

2. $\overline{Channel} = \{ch_0, \cdots, ch_N\} \cup \emptyset$ *is a set of Channels* $ch_i \in \textbf{SharedObject}$ *with* $0 \leq i \leq N$ *and* $N \geq 0$.

3. $\overline{Port} = \{p_0, \cdots, p_N\} \cup \emptyset$ *is a set of External Ports* $p_i \in \textbf{Port}$ *with* $0 \leq i \leq N$ *and* $N \geq 0$. *This set of ports defines communication points to the outside world. All external ports of ALS are connected to the testbench infrastructure that models the behavior of the system's environment. As for hierarchical Behaviors each port can only be bound once. Hence is not allowed to bind a port of* $\overline{Actor}$ *to two different ports in* $\overline{Port}$.

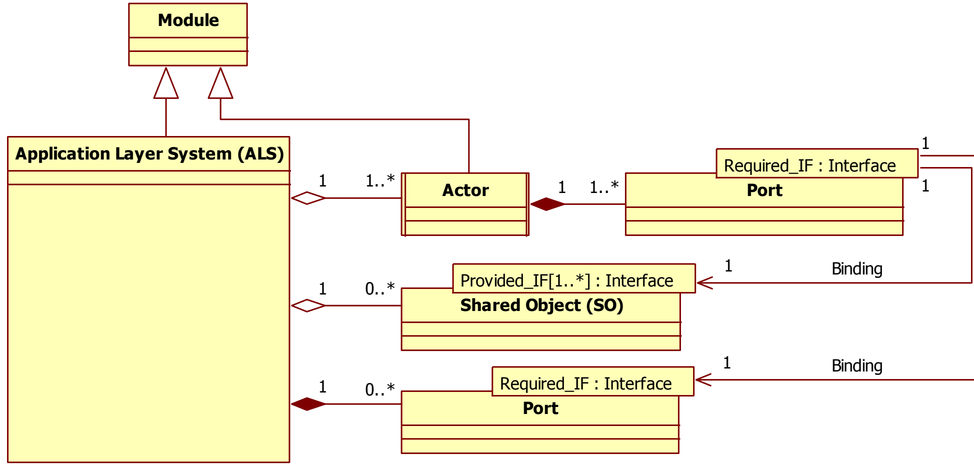


Figure 5.36: Meta-Model of the Application Layer System

4. $\overline{Binding} = \{\triangleright_0, \cdots, \triangleright_N\} \cup \emptyset$ *is a list of binding relations* $\triangleright$ *with*

$$N = \left(\sum_{i=0}^{|\overline{Actor}|-1} |a_i.\overline{Port}| \right) - 1$$

where $a_i \in \overline{Actor}$. *The list of binding relations has the same properties as the binding relation for Behaviors:*

(a) Ports of all child Actors ($\in \overline{Actor}$) *are bound once to either External Ports* ($\in \overline{Port}$) *or interfaces of Channels* ($\in \overline{Channel}$)*:*

$$\forall child_i \in \overline{Actor} \forall p_j \in child_i.\overline{Port} \quad : \quad \exists! \triangleright \in Binding \text{ with } p_j \triangleright p \in \overline{Port} \vee p_j \triangleright ch \in \overline{Channel}$$

(b) One port is uniquely bound to another port. I.e. not two different ports can be bound to the same port:

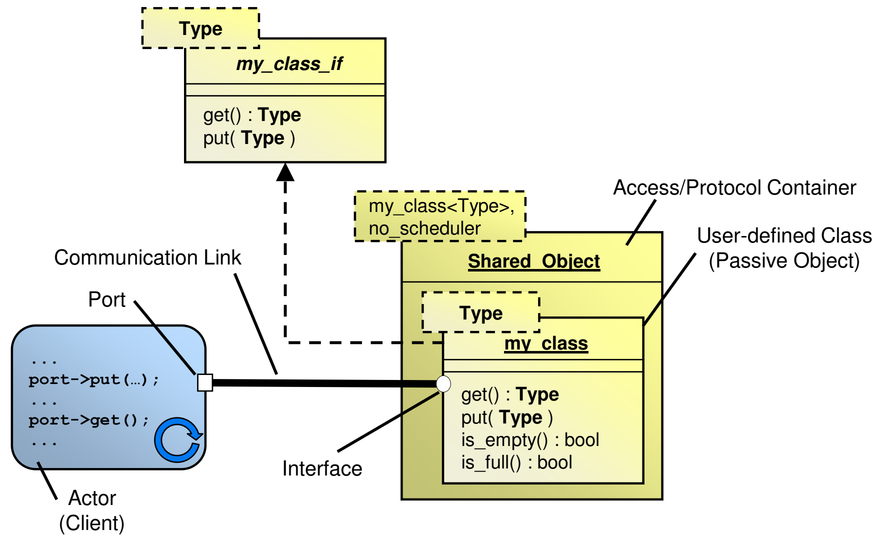
$$\forall \triangleright_i \in Binding, p \in \bigcup_{child \in \overline{Actor}} child.\overline{Port} \quad : \quad \exists! p_j \in \overline{Port} \wedge p \neq p_j \text{ with } p \triangleright_i p_j$$

□

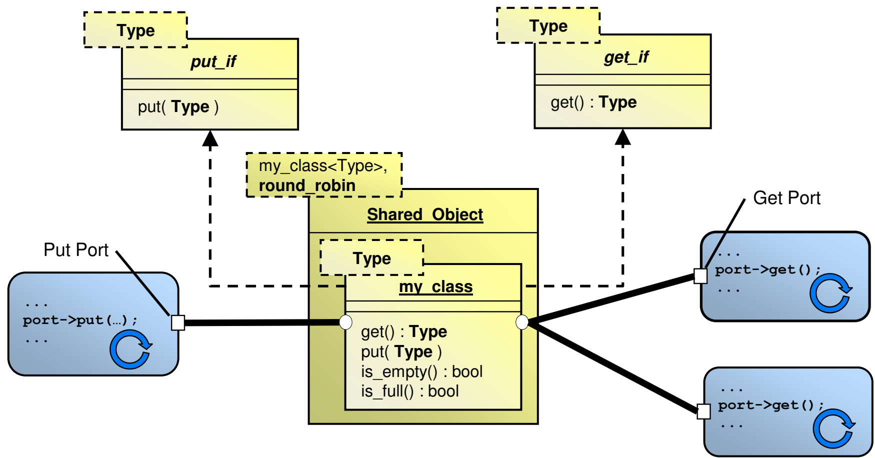
5.5.2.3 Shared Objects

Shared Objects are the modeling primitives for expressing communication between *Actors*. Following the object-oriented programming paradigm, it provides a set of services, grouped by interfaces, to its clients. In our definition a Shared Object offers Services to its clients, which are Actors, that require subsets of the offered Services. In this terminology a Shared Object is a *server* and an Actor or Task is a *client*.

Figure 5.37 gives a graphical overview of two different Shared Object usages. Figure 5.37a shows the user-defined class `my_class` inside a Shared Object. The client accesses the provided services of the `my_class_if` through a port. Since only a single client accesses the Shared Object, no scheduling is required. Figure 5.37b shows the same user-defined class inside a Shared Object. In this configuration `my_class` implements two interfaces `put_if` and `get_if` which are used by three clients. One client uses the `put` interface and two other clients are using the `get` interface. Concurrent accesses are arbitrated using a round robin scheduler.



(a) no scheduling



(b) scheduling

Figure 5.37: Shared Object

In the following sections a definition of the Shared Object is given.

Definition 5.5.2.3 (Shared Object):
An OSSS Shared Object *is a tuple* $SO = [\bar{c}_{parent}, State, Method, Scheduler]$*, where*

1. $\bar{c}_{parent} = \{c_{IF_0}, \cdots, c_{IF_N}, c\}$*, with* $typeof(c_{IF_i}) = IF, 0 \leq i \leq N$ *and* $typeof(c) = C$ *is a list of base classes, consisting of one up to* N *Interface Classes (*IF*), and one or no regular Class (*C*).*

   The function $if_closure$ *collects all valid interface combinations along the interface inheritance relation. It is defined in the following way:*

$$
\begin{aligned}
if_closure(X) &: \quad SO \cup C \rightarrow \overline{IF} \\
if_closure(X) &= \quad \emptyset \\
&\qquad if\ typeof(X) = IF \land X.\bar{c}_{parent} = \emptyset \\
if_closure(X) &= \quad if_closure\left(\downarrow_{IF} (X.\bar{c}_{parent})\right) \ \cup
\end{aligned}
$$

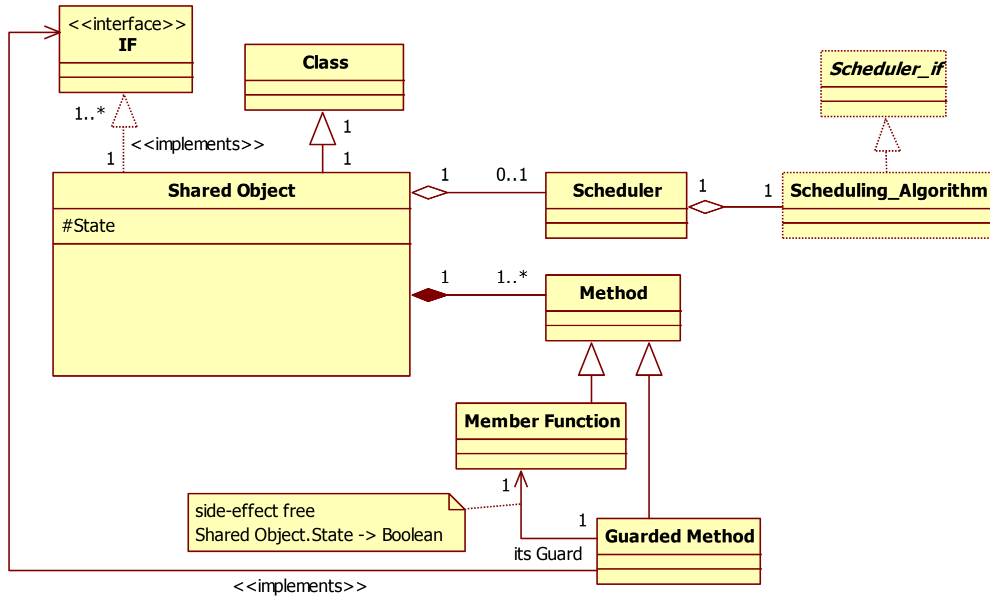


Figure 5.38: Meta-Model of Shared Objects

$$
if_closure(X) = \begin{cases} flat_if\,(\downarrow_{IF}(X.\bar{c}_{parent})) \\ \quad if\ \downarrow_C(X.\bar{c}_{parent}) = \emptyset \land X.\bar{c}_{parent} \neq \emptyset \\ if_closure\,(\downarrow_{IF}(X.\bar{c}_{parent})) \cup \\ if_closure\,(\downarrow_{C}(X.\bar{c}_{parent})) \cup \\ flat_if\,(\downarrow_{IF}(X.\bar{c}_{parent})) \\ \quad if\ \downarrow_C(X.\bar{c}_{parent}) \neq \emptyset \end{cases}
$$

with the projection function:

$$
\begin{aligned}
\downarrow_X(Y) &: \overline{SO} \cup \overline{C} \rightarrow \overline{X},\ X \in \{IF, C\} \\
\downarrow_X(Y) &= \left\{ \bigcup_{i=0}^{|Y.\bar{c}_{parent}|-1} c_i \mid c_i \in Y.\bar{c}_{parent} \land typeof(c_i) = X \right\}
\end{aligned}
$$

Figure 5.39 show an example of the application of the $if_closure$ function to a complex class inheritance relation.

To get the set of Services S a Shared Object provides and implements as Guarded Methods, we need to create the union of all services using the service_closure function:

$$
\begin{aligned}
service_closure(X) &: \overline{IF} \rightarrow \overline{S} \\
service_closure(X) &= \bigcup_{if \in if_closure(X)} if.Method
\end{aligned}
$$

Only the set of Services $S = service_closure(SO)$ can be accessed by clients. All other methods are only accessible inside the Shared Object. For the example in Figure 5.39 $service_closure(F) = \{a, b, c, d, e\}$.

2. *$State = T_0 \times \ldots \times T_n$ is the state vector, with $n \geq 0$, where $T_0, \cdots, T_n$ denote some basic types, classes, or arrays.*
3. *$Method = M \cup GM$ is the set of regular member functions (as defined for classes) $M = \emptyset$ or $M = \{m_0, \cdots, m_m\}$ with $m_i, i \geq 0$, and Guarded Methods $GM = \{gm_0, \cdots, gm_n\}$ with*

$gm_j, j \geq 0$ of the following kinds:

$$
\begin{aligned}
\textbf{\textit{guarded}}\ name &: \langle guard\colon SO.State \rightarrow \textbf{\textit{Boolean}}, \textbf{\textit{void}} \rightarrow \textbf{\textit{void}} \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon \textbf{\textit{true}}, \textbf{\textit{void}} \rightarrow \textbf{\textit{void}} \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon SO.State \rightarrow \textbf{\textit{Boolean}}, \textbf{\textit{void}} \rightarrow T \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon \textbf{\textit{true}}, \textbf{\textit{void}} \rightarrow T \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon SO.State \rightarrow \textbf{\textit{Boolean}}, T_0 \times \cdots \times T_n \rightarrow \textbf{\textit{void}} \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon \textbf{\textit{true}}, T_0 \times \cdots \times T_n \rightarrow \textbf{\textit{void}} \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon SO.State \rightarrow \textbf{\textit{Boolean}}, T_0 \times \cdots \times T_n \rightarrow T \rangle \\
\textbf{\textit{guarded}}\ name &: \langle guard\colon \textbf{\textit{true}}, T_0 \times \cdots \times T_n \rightarrow T \rangle
\end{aligned}
$$

*where each Guarded Method has exactly one associated guard function $guard\colon SO.State \rightarrow$ **Boolean**. A guard is a side-effect free Boolean condition over the state space of the Shared Object $SO.State$. A guarded method is only accessible if the guard function returns **true**. If the guard function instead is defined to be always true, the associated guarded method is always accessible (un-guarded).*

We define $\overline{req}_{SO} \in \textbf{\textit{Integer}}^N_{[0,|GM|-1]}$ as the so-called client request vector. It has the size N of number of clients connected to this Shared Object and contains the identifier of the requested guarded method of each client. Applying an index function index that uniquely assigns indexes to a set S of size $|S| = M$, results in $index(S) = \{s_0, \cdots, s_{M-1}\}$, with $s_i \in S$. Guarded methods can be uniquely numbered using this index function in the following way:

$$GM = index(service_closure(\overline{c}_{parent}))$$

The function $index^{-1}$ returns the index of a given Guarded Method:

$$
\begin{aligned}
index^{-1} &: GM \times \textbf{\textit{integer}}_{[0,|GM|-1]} \rightarrow gm,\ with\ gm \in GM \\
index^{-1}(GM, i) &= gm_i,\ with\ i \in [0, |GM|-1]
\end{aligned}
$$

The function $eval_guard\colon GM \times \textbf{\textit{integer}}_{[0,|GM|-1]} \rightarrow \textbf{\textit{Boolean}}$ evaluates the guard associated to a Guarded Method $gm_i \in GM$, written as $gm_i.guard$:

$$eval_guard(GM, i) = index^{-1}(GM, i).guard$$

With these functions we can define the guard evaluation function guard_eval that transforms the client request vector $\overline{req}_{SO}$ into the guarded request vector $\overline{guarded_req}_{SO} \in \textbf{\textit{Boolean}}^N$ for number of clients N in the following way:

$$
\begin{aligned}
guard_eval &: \textbf{\textit{integer}}^N_{[0,|GM|-1]} \rightarrow \textbf{\textit{Boolean}}^N \\
guard_eval(\overline{req}_{SO}) &= \left\{ \overline{guarded_req}_{SO}[i] = eval_guard(GM, \overline{req}_{SO}[i]) | \forall i \in [0, N-1] \right\}
\end{aligned}
$$

4. *Scheduler is a function $Scheduler\colon \textbf{\textit{Boolean}}^N \rightarrow \textbf{\textit{Integer}}_{[0,N-1]}$, which selects from the guarded request vector $\overline{guarded_req}_{SO}$ the client number $guarded_req_i \in \textbf{\textit{Integer}}_{[0,N-1]}$ to be granted access to the requested guarded method in $\overline{req}_{SO}$.*
The next client that is allowed to access the Shared Object SO can be computed as:

$$next_access(SO) = Scheduler(guard_eval(\overline{req}_{SO}))$$

The associated guarded method to be accessed next is:

$$\overline{req}_{SO}[next_access(SO)]$$

□

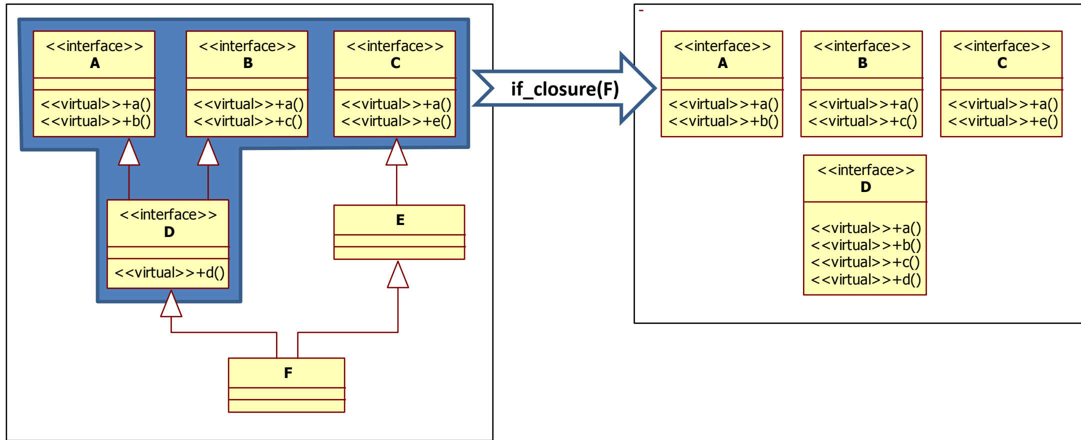


Figure 5.39: Example of $if_closure$ function applied to a mixed class and interface class inheritance. The interface closure of class F $if_closure(F)$ contains all valid combinations of interface classes of class F.

5.5.3 Pre-defined Scheduling Algorithms

Definition 5.5.3.1 (Scheduling):
Scheduling is the method by which concurrent clients (e.g. processes, threads or data flows) are given access to a shared resource (e.g. processor, memory or communication channel). This is usually done to load balance a system effectively or to achieve a certain quality of service. A scheduling algorithm defines access rules for all clients to the shared resource (also called server). Scheduling algorithms can be specified to target the following objectives:

Throughput *The total number of clients that complete their execution per time unit*

Latency *more specifically:*

- ***Turnaround time:*** *total time between submission of a client's request to use the shared resource and its completion*
- ***Response time:*** *amount of time it takes from when a client's request was submitted until the first shared resource's response is produced*

Waiting Time *The time each client (or more generally appropriate times according to each clients' priority) is waiting/blocked until access to the shared resource is granted*

Fairness *The same amount of accesses or the same waiting times of all clients (or more generally: proportional to each clients' priority)*

In practice, these objectives often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs. □

In this work, scheduling will be used to serialize concurrent accesses to *Shared Objects* and *Communication Elements* of the Virtual Target Architecture Layer (see Section 5.6.2). To enable reuse and plug-and-play of different scheduling algorithms the following generic scheduling interface is provided.

Definition 5.5.3.2 (Scheduler Interface):
Given a shared resource S and a number of $N > 1$ clients applying for accessing S. The function $Scheduler_{IF}: \textbf{Boolean}^N \rightarrow \textbf{Integer}_{[0,N-1]}$ defines the Scheduler Interface for a shared resource S.

*The scheduler interface gets a request vector $\overline{request} \in \textbf{Boolean}^N$ that is a Boolean Array of size N. Each client that is applying for access to S issues its request by assigning **true** to its unique position in the array. Each client's position in the request vector depends on its priority.*

The interpretation of the priority index ("zero is highest" or "zero is lowest" priority) is user defined and needs to be handled by the scheduling algorithm appropriately. After applying the scheduling algorithm on the given request vector $\overline{request}$ *the client at position* $i \in \boldsymbol{Integer}_{[0,N-1]}$ *gets access to the shared resource* S. □

Some frequently used scheduling algorithms are available, but also user-defined schedulers can be specified using the generic scheduler interface described above. Pre-defined scheduling algorithms are described in the following paragraphs.

5.5.3.1 Static Priority

The static priority scheduling algorithm picks the requesting client with the highest priority from the request vector $\overline{request}$ and returns its index (see Algorithm 1). Depending on the interpretation of priorities ("zero is highest" or "zero is lowest") the request vector is traversed in forward or backward direction (default: zero is lowest priority).

This algorithm does not guarantee fairness, because priorities are static. Depending on the request characteristics of the clients it may happen that high priority clients get much more often access to the shared resource than low priority clients. In some cases low priority clients might get never get access to the shared resource at all.

Algorithm 1 Static Priority Scheduler (with N number of clients)

```
1:  if zero_is_highest = true then
2:    for i = 0 → (N − 1) do
3:      if request[i] then
4:        return i
5:      end if
6:    end for
7:  else
8:    for i = 0 → (N − 1) do
9:      if request[N − 1 − i] then
10:       return N − 1 − i
11:     end if
12:   end for
13: end if
```

5.5.3.2 Ceiling Priority

The ceiling priority scheduling algorithm (see Algorithm 2) all clients are initially sorted in a certain order, which cannot be influenced and must be regarded as unknown. This order can be seen as the initial priority of the clients, with the client having the lowest index having the highest priority. Whenever the schedule function is invoked, the requesting client with the highest priority is granted, and it is assigned the lowest priority.

Other than in the static priority algorithm, the scheduled client's priority is dynamically changed to the lowest possible priority (see lines 14-16). This way the ceiling priority algorithm guarantees fairness.

The initialization in lines 2-4 is executed only once at the beginning. It is skipped in further scheduler calls.

5.5.3.3 Round Robin

In the round robin scheduling algorithm (see Algorithm 3) all clients are sorted in a fixed order and numbered from 0 to N (where $N-1$ is the number of clients). If client m, with $0 \leq m \leq N$ was granted last, the first requesting client following in that order will be granted next. If no client i, with $m < i \leq N$ is requesting, counting starts again from 0 (i.e., clients are treated as being organized in a logical ring). This algorithm requires no client priorities. In the case of clients with priorities this algorithm simply omits them.

Algorithm 2 Ceiling Priority Scheduler (with N number of clients)

1: << **once only** >>
2: **for** $i = 0 \rightarrow (N-1)$ **do**
3:   **Integer**$_{[0,N-1]}$ history[i] $\leftarrow i$
4: **end for**
5:
6: **Integer**$_{[0,N-1]}$ grant $\leftarrow 0$
7: **Boolean** ripple $\leftarrow$ **false**
8: **for** $i = 0 \rightarrow (N-1)$ **do**
9:   **if** $\overline{request}$[history[i]] $\wedge \neg$ripple **then**
10:     grant $\leftarrow$ history[i]
11:     ripple $\leftarrow$ **true**
12:   **end if**
13:   **if** ripple $\wedge$ $(i \neq (N-1))$ **then**
14:     **Integer**$_{[0,N-1]}$ swap $\leftarrow$ history[i]
15:     history[i] $\leftarrow$ history[$i+1$]
16:     history[$i+1$] $\leftarrow$ swap
17:   **end if**
18: **end for**
19: **return** grant

The initialization in line 2 is executed only once at the beginning. It is skipped in further scheduler calls.

Algorithm 3 Round Robin Scheduler (with N number of clients)

1: << **once only** >>
2: **Integer**$_{[0,N-1]}$ last_grant $\leftarrow N-1$
3:
4: **Integer**$_{[0,N-1]}$ next_grant $\leftarrow$ last_grant
5: **Boolean** break $\leftarrow$ **false**
6: **for** $i = 0 \rightarrow (N-1)$ **do**
7:   **if** $\neg$break **then**
8:     **if** next_grant $= (N-1)$ **then**
9:       next_grant $\leftarrow 0$
10:     **else**
11:       next_grant $\leftarrow$ next_grant $+ 1$
12:     **end if**
13:     **if** $\overline{request}$[next_grant] **then**
14:       break $\leftarrow$ **true**
15:     **end if**
16:   **end if**
17: **end for**
18: last_grant $\leftarrow$ next_grant
19: **return** next_grant

5.5.3.4 Modified Round Robin

The modified round robin scheduling algorithm (see Algorithm 4) works in almost the same manner as the round robin algorithm above. All clients are sorted in a fixed order and numbered from 0 to N (where $N-1$ is the number of clients). If the scheduling method is activated, the first requesting client following in that order onto a certain start index m, with $0 \leq m \leq N$, is granted next. Each time a client was granted the start index m is increased by 1, and, if $m+1$ exceeds N, reset to 0 (i.e., clients are treated as being organized in a logical ring). This algorithm requires no client priorities. In the case of clients with priorities this algorithm simply

omits them.

The initialization in line 2 is executed only once at the beginning. It is skipped in further scheduler calls.

Algorithm 4 Modified Round Robin Scheduler (with N number of clients)

```
1:  << once only >>
2:  Integer[0,N−1] last_grant ← N − 1
3:
4:  Integer[0,N−1] next_grant ← last_grant
5:  Boolean break ← false
6:  for i = 0 → (N − 1) do
7:    if ¬break then
8:      if next_grant = (N − 1) then
9:        next_grant ← 0
10:     else
11:       next_grant ← next_grant + 1
12:     end if
13:     if request[next_grant] then
14:       break ← true
15:     end if
16:   end if
17: end for
18: if last_grant ≠ next_grant then
19:   if last_grant = (N − 1) then
20:     last_grant ← 0
21:   else
22:     last_grant ← last_grant + 1
23:   end if
24: end if
25: return next_grant
```

5.5.4 Timing Annotations

Code blocks within Actors and Shared Object services and guards can be annotated by Estimated Execution Times (EET) and Required Execution Times (RET, monitored during simulation) as shown in Figure 5.40. With these timing annotations, early performance validation can be performed.

5.5.4.1 Shared Object annotations

Figure 5.40 depicts a bounded size FIFO (similar to the Queue Channel of the Behavior Layer) encapsulated by a Shared Object. The interface definitions for putting and getting items of data type `T` to/from the FIFO is shown in Listing 5.11. These interface methods of the Shared Object, also called services, can be called by Actors connected through ports of the proper interface type. The FIFO class implementing these interfaces as guarded methods is show in Listing 5.12.

The `put` method is guarded by the `!is_full()` side-effect free guard. It takes care that the `put` is blocked when the FIFO is full. I.e. `put` can only be called when the FIFO is not full.

The `get` method is guarded by the `!is_empty()` side-effect free guard. This way `get` can only be called when there is at least a single item in the FIFO, otherwise a call of `get` is blocked until an item has been written to the FIFO.

Both, guarded methods and guards have been annotated with the following timing labels:

$FIFO.EET_{put} :=$ `t_put`

$FIFO.EET_{get} :=$ `t_get`

$FIFO.EET_{g0} :=$ `t_g0`

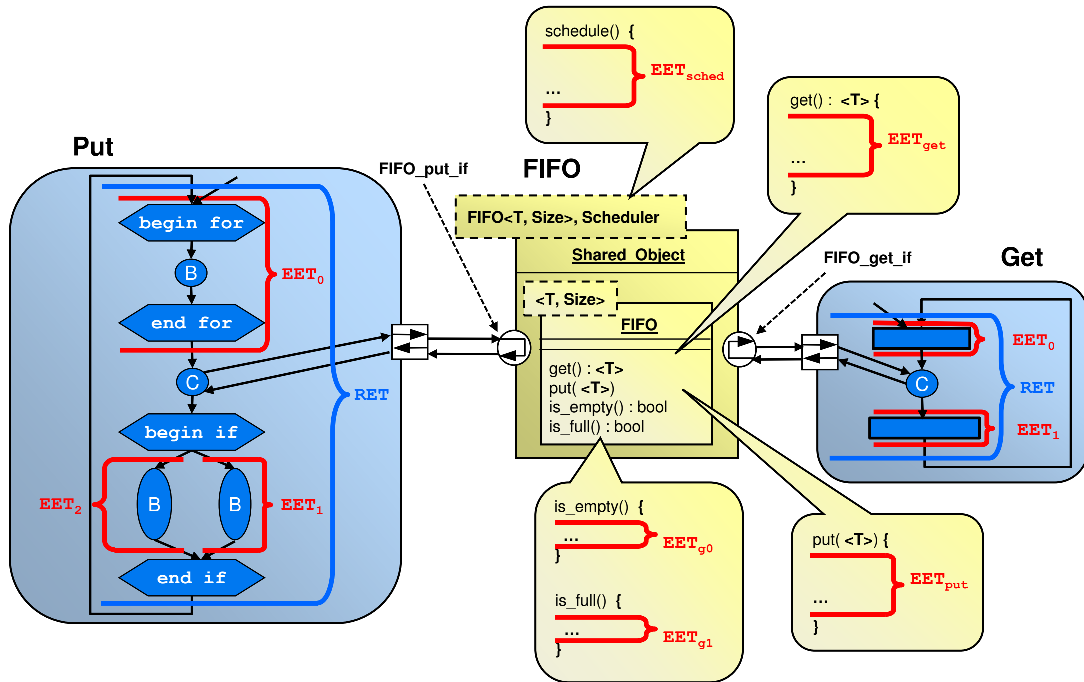


Figure 5.40: Timing Annotations at the Application Layer

```
1  template<class T>
2  class FIFO_put_if : public virtual sc_interface {
3  public:
4    virtual void put(T item) = 0;
5  };
6
7  template<class T>
8  class FIFO_get_if : public virtual sc_interface {
9  public:
10   virtual T get() = 0;
11 };
```

Listing 5.11: FIFO put and get interface, see Figure 5.40

$FIFO.EET_{g1}$:= `t_g1`

5.5.4.2 Actor annotation

As defined in 5.5.2.1 an Actor consists of a (hierarchical) Behavior. Depending on the allowed set of behavior compositions actors have been classified into *Active Sequential, Active Parallel, Reactive Sequential and Reactive Parallel.* Independently from this classification, timing annotations are performed in leaf behavior's `main` routines only. Timing annotations can be done at different granularities, ranging from complete functions over basic blocks to single statements, provided that the following rules are fulfilled:

- timing block annotations cannot be nested (i.e. an EET block is not allowed to contain another EET block)
- timing block annotations cannot be overlapping (i.e. an EET block must be completed before an new EET block begins)
- calls on communication ports (to Shared Objects) are not allowed to be inside any EET block annotation

```
template<class T, unsigned int Size>
class FIFO : public FIFO_put_if<T>, public FIFO_get_if<T> {
 public:
  FIFO() : m_put_index(0), m_get_index(0), m_num_items(0) {}

  OSSS_GUARDED_METHOD_VOID(put, OSSS_PARAMS(1, T, item), !is_full() ) {
    OSSS_EET(t_put) {
      m_buffer[m_put_index] = item;
      if (m_put_index == (Size-1)) m_put_index = 0;
      else m_put_index += 1;
      m_num_items += 1;
    }
  }

  OSSS_GUARDED_METHOD(T, get, OSSS_PARAMS(0), !is_empty() ) {
    OSSS_EET(t_get) {
      T result = m_buffer[m_get_index];
      if (m_get_index == (Size-1)) m_get_index = 0;
      else m_get_index += 1;
      m_num_items -= 1;
      return result;
    }
  }

  bool is_empty() const {
    OSSS_EET(t_g0) {
      return m_num_items == 0;
    }
  }

  bool is_full() const {
    OSSS_EET(t_g1) {
      return m_num_items == Size;
    }
  }

 protected:
  unsigned int m_put_index, m_get_index, m_num_items;
  T m_buffer[Size];
};
```

Listing 5.12: FIFO class with implementation of guarded put and get methods with timing annotations, see Figure 5.40

In contrast, RETs can be arbitrarily nested, contain calls on communication ports but may not overlap.

Listing 5.13 shows RET and EET annotations inside the Active Sequential Put Actor, as visualized in Figure 5.40. The following timing labels have been used:

$Put.RET :=$ `t`

$Put.EET_0 :=$ `t_0`

$Put.EET_1 :=$ `t_1`

$Put.EET_2 :=$ `t_2`

Listing 5.14 shows RET and EET annotations inside the Active Sequential Get Actor. Here the following timing labels have been used:

$Get.RET :=$ `t`

$Get.EET_0 :=$ `t_0`

$Get.EET_1 :=$ `t_1`

```
1  class Put : public osss_actor {
2  public:
3    osss_port<osss_shared_if< FIFO_put_if<Packet> > > p_out;
4
5    OSSS_ACTOR_CTOR(Put) { }
6
7    void main() {
8      Packet p;
9      unsigned int c = 0;
10     while(true) {
11       OSSS_RET(t) {
12         OSSS_EET(t_0) {
13           for(int i=0; i<p.size()-1; i++) {
14             p.set_payload(i,c);
15             c++;
16           }
17         }
18
19         p_out->put(p); // Shared Object service call
20
21         if (c>=42) { OSSS_EET(t_1) { c = c*c; } }
22         else { OSSS_EET(t_2) { c = 0; } }
23       }
24     }
25   }
26 };
```

Listing 5.13: Put Actor implementation with timing annotations, see Figure 5.40

```
1  class Get : public osss_actor {
2  public:
3    osss_port<osss_shared_if< FIFO_get_if<Packet> > > p_in;
4
5    OSSS_ACTOR_CTOR(Get) { }
6
7    void main() {
8      Packet p;
9      unsigned int sum = 0;
10     while(true) {
11       OSSS_RET(t) {
12         OSSS_EET(t_0) { for(int i=0; i<p.size()-1; i++) { p.set_payload(i,0); } }
13
14         p = p_in->get(p); // Shared Object service call
15
16         OSSS_EET(t_1) { for(int i=0; i<p.size()-1; i++) { sum += p.get_payload(i); } }
17       }
18     }
19   }
20 };
```

Listing 5.14: Get Actor implementation with timing annotations, see Figure 5.40

5.5.4.3 Timing estimation

EETs are obtained from a processing element dependent timing estimation. In literature different approaches have been presented so far. An overview of these approaches (including power estimation) and a reference framework for timing and power back annotation has been presented in [12, 5]. At the Application Layer we are interested in timing estimations for Actors running on programmable processing elements (i.e. Actor software implementation) and custom hardware processors (i.e. Actor hardware implementation). In this work the Shared Object can only by implemented as custom hardware processor.

In Figure 5.41 the process of execution time estimation and back-annotation for programmable processing elements is visualized on three different levels:

Execution Time Estimation as shown in Figure 5.41a enables the timing annotation of basic

blocks of each Leaf Behavior of an Actor. The basic blocks are obtained from the Control Data Flow Graph (CDFG) generated by a C/C++ compiler front-end. With a microarchitecture model of the targeted processing element the data flow of the basic blocks can be scheduled. With the cycle time of each instruction the delay of each basic block can be computed.

Processor Timing Estimation adds more microarchitecture specific timing information. For simple micro-controllers the naive execution time estimation as described above works well, but for pipelined processors the microarchitectural pipeline scheduling, external memory delay and branch prediction delay need to be considered, too. Figure 5.41b shows the example of simple in-order, single issue processor with a three stage pipeline (Instruction Fetch, Instruction Decode and Execute). The processor datapath model consists of a single Arithmetic Logical Unit (ALU) with a latency of one clock cycle for each operation. In this processor datapath model we assume no cache (or a 100% cache hit rate). For the intermediate 3-address code the operation delay after pipeline scheduling is 42.

Stochastic Memory Delay Model adds data and instruction memory access overhead and branch prediction delay. For using a statistical model the cache and branch prediction hit rates need to be available in the data model. In Figure 5.41c a direct-mapped cache of size 16K with a cache delay $CD = 1$ cycle, an instruction hit rate $HR_I = 97.79\%$ and a data hit rate $HR_D = 69.96\%$ is used. The branch predictor has a miss penalty rate $MP_{rate} = 60\%$ and a penalty $BP = 2$ cycles. The latency of the external memory (to refresh the cache) is $L_{mem} = 8$ cycles. With these parameters the following simple statistical memory delay model can be used:

$$\begin{aligned} \text{operation access overhead} &= N_{op} \cdot ((1.0 - HR_I) \cdot (CD + L_{mem})) \\ \text{data access overhead} &= N_{ld} \cdot ((1.0 - HR_D) \cdot (CD + L_{mem})) \\ \text{branch prediction overhead} &= MP_{rate} \cdot BP \end{aligned}$$

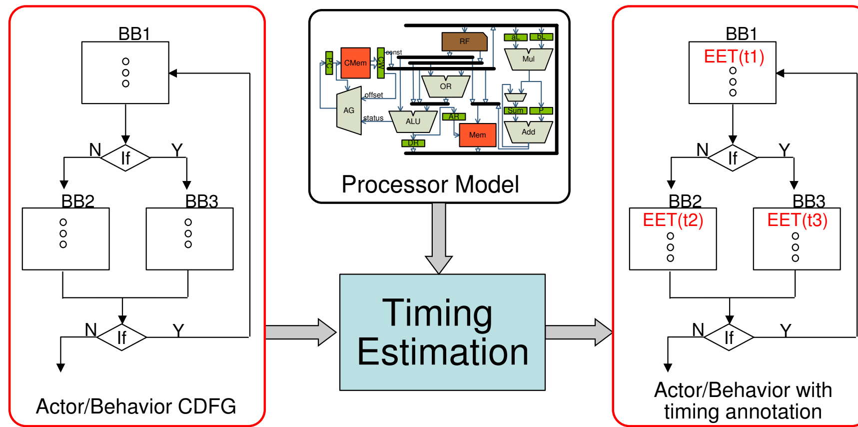
N_{op} is the number of operations and N_{ld} is the number of data accesses (load and store) per basic block. With the given cache and branch prediction model, the memory access overhead (operation and data access) is 4.1 and the branch delay is 1.2. With this overhead the EET for the intermediate 3-address code example is $42 + 4.1 + 1.2 = 47.3$ cycles. In this case the timing annotation would result in $47.3 \cdot 1/f_{clk}$ seconds.

5.5.4.4 Timing analysis

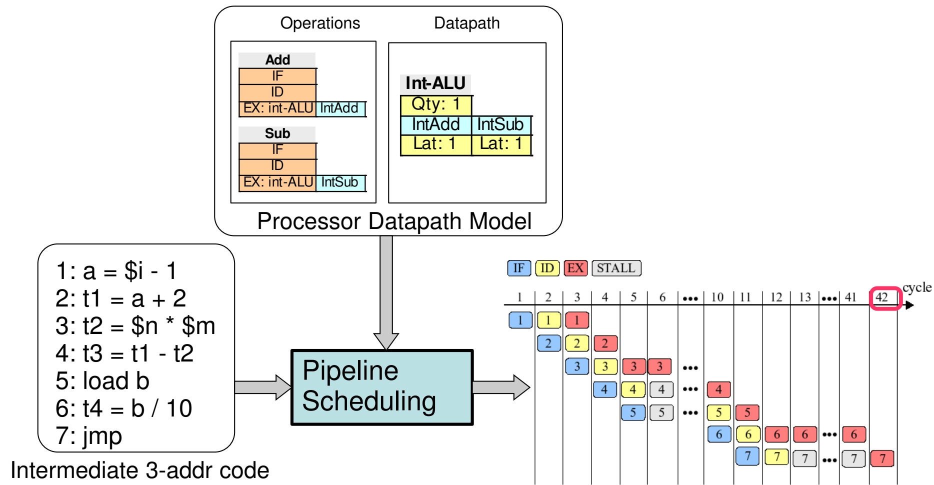
In the FIFO example shown in Figure 5.40 both actors are executed cyclically. The *Put.RET* and *Get.RET* describe the deadlines of the Put and Get Actors. Depending on the timing annotations of the Actors and the FIFO Shared Objects, an upper boudn to the *Put.RET* and *Get.RET* can be calculated as:

$$\begin{aligned} Put.RET &\leq Put.EET_0 + T_{blocking}(Put) + T_{GE} + FIFO.EET_{sched} + FIFO.EET_{put} + \\ &\quad \max(Put.EET_1, Put.EET_2) \\ Get.RET &\leq Get.EET_0 + T_{blocking}(Get) + T_{GE} + FIFO.EET_{sched} + FIFO.EET_{get} + \\ &\quad Get.EET_1 \end{aligned}$$

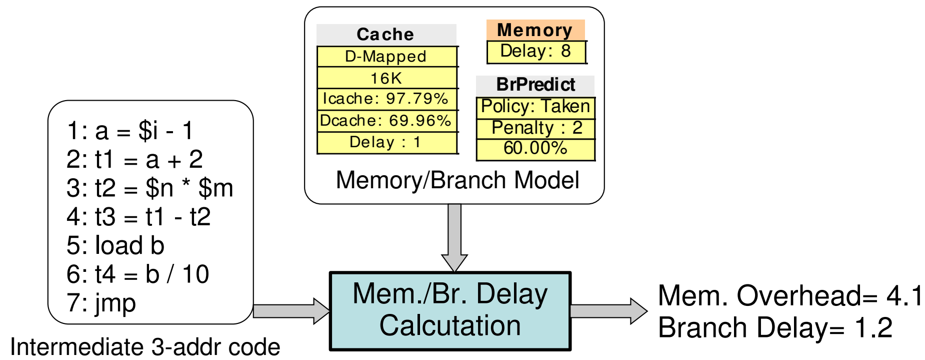
where $T_{GE} \leq NC \cdot \max(FIFO.EET_{g0}, FIFO.EET_{g1})$ is the upper timing bound of the guard evaluation depending on the size of the “request vector” of size NC (which is the total number of clients connected to the Shared Object). The blocking time $T_{blocking}$ is a more complex scheduling algorithm and guarded method dependent parameter. If the “wrong” scheduling algorithm (e.g. a scheduling algorithm that is not fair) in combination with unguarded or only partially guarded methods is chosen $T_{blocking}$ can become ∞. However, ∞ blocking times are considered as design error. To avoid this kind of livelock, proper analysis of the implemented Shared Object guarded methods in combination with scheduling algorithm needs to be analyzed. In Section 5.5.6 timed automata will be used to check this kind of property for Shared Objects.



(a) Execution Time Estimation



(b) Processor Timing Estimation



(c) Stochastic Memory Delay Model

Figure 5.41: Processing element timing estimation and back-annotation [27]

In the bounded FIFO example from Figure 5.40 a scheduling independent upper bound for $T_{blocking}$ can be calculated as

$$
\begin{aligned}
T_{blocking}(Put) &\geq 0 \\
T_{blocking}(Put) &\leq Size \cdot (Get.EET_0 + T_{GE} + FIFO.EET_{sched} + FIFO.EET_{get} + \\
&\quad Get.EET_1) \\
T_{blocking}(Get) &\geq Put.EET_0 + T_{GE} + FIFO.EET_{sched} + FIFO.EET_{put} + \\
&\quad \max(Put.EET_1, Put.EET_2) \\
T_{blocking}(Get) &\leq Size \cdot (Put.EET_0 + T_{GE} + FIFO.EET_{sched} + FIFO.EET_{put} + \\
&\quad \max(Put.EET_1, Put.EET_2))
\end{aligned}
$$

5.5.4.5 Properties of Timing Annotations

EET (Estimated Execution Time) represents the passing of time as a duration D which is defined as $D = (value, unit)$, with the $value \in \mathbb{R}_0^+$ that specifies the duration, and its associated time $unit \in \{fs, ps, ns, \mu s, ms, s\}$, with the the following meaning

$$
\begin{aligned}
|X| &: unit \to \mathbb{R}^+ \\
|fs| &= 10^{-15} \quad \text{(femto second)} \\
|ps| &= 10^{-12} \quad \text{(pico second)} \\
|ns| &= 10^{-9} \quad \text{(nano secons)} \\
|\mu s| &= 10^{-6} \quad \text{(micro second)} \\
|ms| &= 10^{-3} \quad \text{(milli second)} \\
|s| &= 10^{0} \quad \text{(second)}
\end{aligned}
$$

EETs can have the following orthogonal properties

- **scalar** or **duration interval**: the specified duration of an EET annotation can either be a simple scalar, e.g. $d = (42, \mu s)$, or it can be an interval with a lower $D_l \in D$ and upper $D_u \in D$ bound $[D_l, D_u]$. This can be used to specify a best-case to worst-case interval. Associating this with a Probability Density Function (PDF, see Definition 5.5.4.1) enables a probabilistic weighting of duration occurrences in the given duration interval.

  If not specified for a duration interval, by default, we use the following uniform distribution for the duration interval $[a = D_l.value \cdot |D_l.unit|, b = D_u.value \cdot |D_u.unit|]$, $a < b$ is given:

  density function

  $$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

  distribution function

  $$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$$

- **inline** or **external**: refer both to the location of the timing annotation. Inline annotations are of the behavior description inside the code of the leaf behaviors. External annotations are in a separate location outside the code of the leaf behaviors. In external annotation mode the annotation specifies a unique identifier consisting of the hierarchical instance name of the leaf behavior and a uniquely specified annotation name. This annotation identifier can be used in an external timing configuration file to assign execution durations or intervals. The advantage of the external annotation is that changing execution times does not affect changes in the design. Moreover, different annotation files can be used

to apply different execution time annotations for the same behavior model. A slight disadvantage of the external over the internal annotations is a run-time overhead due to a more time consuming look up during simulation. Reading the external configuration file before and caching it efficiently before simulation starts makes this overhead negligible.

- **block** or **label**: a block annotation has an opening { and a closing } bracket. EET block annotations may not be nested. I.e. an EET block may not contain another EET block. Block timing annotations may be inline or external. In contrast to block annotation a label annotation does not refer to a code block in the leaf behavior, but only specifies the passing of a certain duration or duration interval (between two labels) in a specific code location.

Definition 5.5.4.1 (Probability Density Function (PDF)):
A Probability Density Function (PDF) is most commonly associated with absolutely continuous univariate distributions. A random variable X has density f, where f is a non-negative Lebesgue-integrable function, if:

$$P[a \le X \le b] = \int_a^b f(x)\ \mathrm{d}x = F(b) - F(a)$$

$$\int_{-\infty}^{\infty} f(x)\ \mathrm{d}x = 1$$

Hence, if F is the cumulative distribution function of X, then:

$$F(x) = \int_{-\infty}^{x} f(u)\ \mathrm{d}u,$$

and (if f is continuous at x)

$$f(x) = \frac{\mathrm{d}}{\mathrm{d}x} F(x).$$

□

RET (Required Execution Time) represents a timing requirement to be checked. This timing requirement can be specified as a duration D or duration interval $[D_l, D_u]$ as defined for EETs. RETs can have the following orthogonal properties:

- **scalar** or **duration interval**: a scalar RET duration D has the meaning "not longer than D", while a duration interval has the meaning "not earlier than D_l and not later than D_u. Every scalar duration D can can also be expressed by the interval duration $[0, D]$.
- **inline** or **external**: has exactly the same meaning as for EETs.
- **block** or **label**: Since RETs are used to check a path of EETs, an RET needs to contain at least a single EET. Moreover, and in contrast to EETs, RETs can be nested. RET blocks are specified in the same way as EETs. RET labels can be compared by their identifier. Like in external annotations, the statement annotation is specified by a unique identifier consisting of the hierarchical instance name of the leaf behavior and a uniquely specified annotation name. In the external timing configuration file a duration or duration interval between two labels which need to be checked can be specified.

5.5.4.6 Limitations

Since the Application Layer abstracts from some important target architecture details, which have an impact on the timing of the application, and therefore the accuracy of the application layer timing estimation. The following restrictions apply to EETs:

Pipeline and Data/Instruction fetch Penalties EET annotations are based on a performance model of the computational resources the Actors and Shared Objects are mapped to. If this computation resource is a complex processing element with an internal pipeline (multiple

issue, out of order execution, branch prediction, etc.), data and/or instruction cache all effects related the pipeline delays and cache misses need to be statistically modeled within the timing boundaries of each EET's BCET-WCET interval. Each EET's boundaries could be obtained through static code analysis using state-of-the art WCET tools. The statistical variation of each EET block cannot be obtained automatically so far.

Shared Memory Access Penalties If the data and/or instruction memory (directly connected to the processor or behind a cache) is shared by multiple processors through a crossbar or shared bus, the access penalties caused by blocking times due to concurrent memory accesses are not considered here and beyond the scope of this work. In Section 5.6 considered architectures will be restricted to exclusive data and instruction memories for all processing elements.

Method Call Overhead All timing related penalties for method call to a Shared Object are neglected in Application Layer models. The communication overhead for a method call depends on the chosen communication protocol and spans over the entire protocol stack, from the application layer down to the physical layer. Communication element allocation and communication link binding to communication elements is part of the Application Layer to Virtual Target Architecture mapping. The method call overhead will be modeled on the Virtual Target Architecture Layer in Section 5.6.

Data Transport Overhead relates to the lower layers of the method call communication protocol stack and depends on the chosen communication element and the protocol characteristics. The amount of data to be transported per method call depends on the total size of the method parameters, return value and, encoding/packing and transport/synchronization protocol overhead. These information and associated timing penalties will also be added on the Virtual Target Architecture Layer in Section 5.6.

5.5.5 Mapping rules

For transforming the Behavior Layer model into an Application Layer model, the following steps should be applied:

1. **Actor and Shared Object allocation:** In contrast to the Behavior Layer, the Application Layer explicitly models (shared) computation resources. For modeling computation resources the Actor and Shared Object container classes are provided. Before mapping Behaviors and Channels to these containers, the total number of available Actors and Shared Objects need to be decided by the designer. When targeting a platform FPGA (as assumed in this work), each used software processing element corresponds to the allocation of one Actor per processor. For the remaining space (LUTs and BRAMs) in the FPGA, an Actor for each functionality to be realized as custom hardware needs to be allocated.

   Computation containers representing software processors can be Actors of kind: *Active Sequential* or *Reactive Sequential*.

   Computation containers representing custom hardware processor can be Actors of kind: *Active Parallel* or *Reactive Parallel*.

   Shared Objects are allocated for the communication between Actors (depending on the communication dependencies in the Behavior Layer model) and for modeling shared functionality to be realized in dedicated hardware. Compared to the classification of Sequential and Parallel Actors, Shared Objects are always of kind *Reactive Sequential*.

2. **Behavior partitioning and mapping:** After allocation of computation resources, now the Leaf Behaviors are mapped onto Actors and Shared Objects. Before actually mapping

Behaviors, a partitioning step is performed on the Behavior Layer Model. The goal of this partitioning is to obtain:

(a) A set of parallel scheduled top-level Behaviors, each of them representing a single computational resource of the Application Layer.
(b) Communication Channels between parallel scheduled top-level Behaviors using Behavior Layer Channels with message passing semantics only.

This partitioning step is very similar to the Behavior partitioning in SpecC as described in [27]. The main difference is the handling of synchronization between parallel and pipelined Behaviors running on different computation resources. In SpecC blocking message passing channels (e.g. Double Handshake Channels) are used to retain to fork-join semantics of parallel scheduled Behaviors on different computation resources (called processing elements in SpecC). We are using dedicated Shared Objects to implement the fork-join semantics of `PAR` Behaviors and the pipeline semantics of `PIPE` Behaviors.

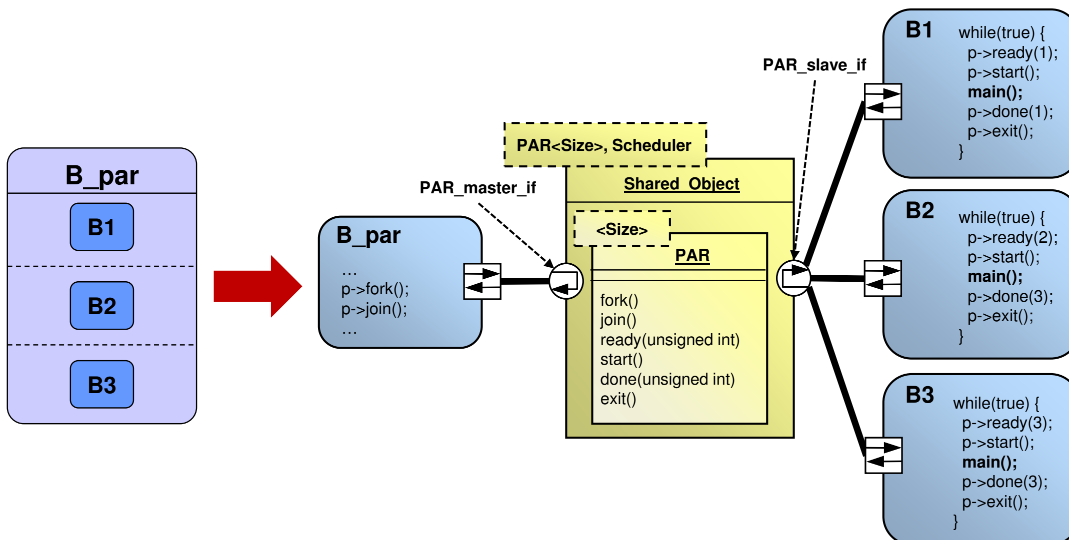


Figure 5.42: `PAR` Behavior to Application Layer transformation

Figure 5.42 depicts the `PAR` Behavior to Application Layer transformation using a dedicated Shared Object to implement the fork-join synchronization of Behaviors `B1`, `B2` and `B3`. The transformation step results in four Actors: one representing the `B_par` composite Behavior, and one for each of the parallel composed Behaviors.

Listing C.6 show the details of the `PAR` Shared Objects used to implement the fork-join semantics. The `PAR_master_if` provides services to initiate the `fork` and to wait for the completion of all forked processes (`join`). The `PAR_slave_if` provides services to each of the forked Actors. When each Actor is `ready` the execution of the `main` routine starts. After execution of its `main` routine, each Actor notifies completion (`done`) and waits for each of the other Actors to finish (`exit`).

Figure 5.43 depicts the `PIPE` Behavior to Application Layer transformation using a dedicated Shared Object to implement the pipeline scheduling of Behaviors `B1`, `B2` and `B3`. The transformation step results in four Actors: one representing the `B_pipe` composite Behavior, and one for each of the pipelined Behaviors.

Listing C.7 show the details of the `PIPE` Shared Objects used to implement the pipeline execution semantics.

The `PIPE_master_if` provides a service to set an upper bound of the execution of each pipeline stage. By default it is zero, resulting in an infinite execution of pipeline. The `fork` service starts the pipeline execution and, if the pipeline execution is bounded, `join` waits for the completion of the pipeline execution, similar to the `PAR` Shared Object.

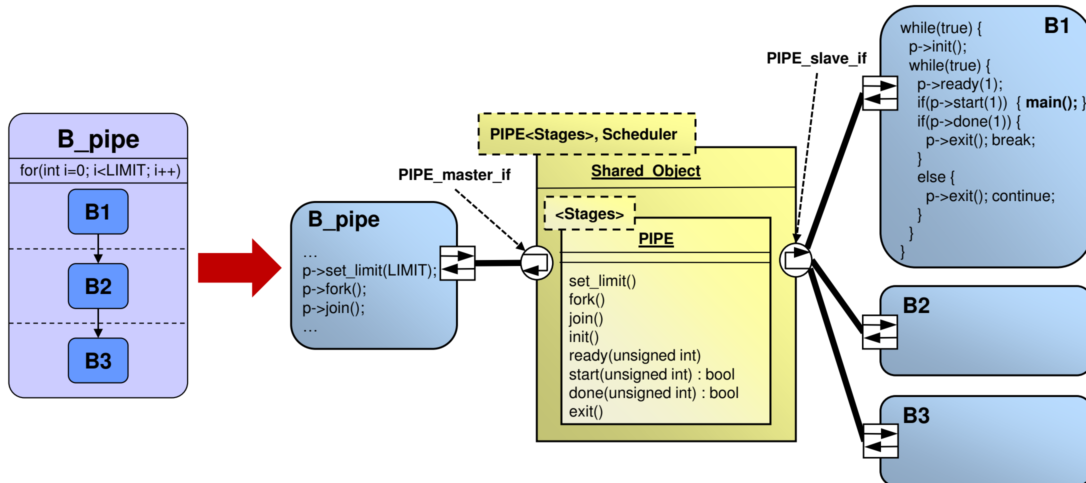


Figure 5.43: `PIPE` Behavior to Application Layer transformation

The `PIPE_slave_if` provides services to each of the pipeline stage Actors. The `init` barrier is taken after the pipeline master has started the pipeline execution through `fork`. When each pipeline stage is `ready` the execution of the `main` routine starts only if the ramp-up phase for the corresponding stage has been completed. At the end of each pipeline stage's execution cycle the `done` call notifies completion of the cycle and checks if the bounded number of executions for each stage has been reached. If this should be the case, the `exit` join-barrier is taken and the `init` barrier is entered again, waiting for the next pipeline activation. If the execution bound has not been reached or the pipeline execution is unbounded, the `exit` barrier is taken and the `ready` service is executed again.

The Behavior Layer `PAR` and `PIPE` to Application Layer Actor and Shared Object transformation as described above is preformed along the Behavior Layer model's hierarchy, resulting in a tree/network of Actors and `PAR`/`PIPE` Shared Objects. During this process new actors are created. If the number of resulting Actors exceeds the limit of allocated Actor resources, pipeline executions can be easily transformed into a sequential execution. Also parallel scheduled Behaviors can be scheduled into a sequential execution sequence. In both cases, communication between formerly parallel scheduled Behaviors requires to be reconsidered. Synchronization dependencies through Handshake and Double Handshake channels need to be properly resolved during rescheduling. In this work we are not going further into details of rescheduling. In [7, 1] more details about Behavior Layer model scheduling, in particular dynamic and communication aware scheduling of parallel Behaviors, is described.

For sequential `SEQ` and finite state machine `FSM` scheduled Behaviors no further adoption is required when integrating these Behaviors into Actors.

3. **Behavior Channel partitioning and mapping:**

   In a last step the remaining Behavior Layer Channels of kind *Shared Variable*, *Piped Variable*, *Queue*, *Handshake* and *Double Handshake* are replaced by their Shared Object implementations (see Appendix C).

5.5.6 Operational Semantics

As well as for the Behavior Layer, the operational semantics of the Application Layer is expressed using timed automata in Uppaal. The timed automata representation expresses the communication protocol between Actors and Shared Objects as well as the timing annotations presented in Section 5.5.4. In this timed automaton model EETs are expressed as timing intervals $[BCET, WCET]$ with best-case and worst-case execution time boundaries. In Uppaal we are using `clock` variables in combination of invariants and guard to express the EET intervals.

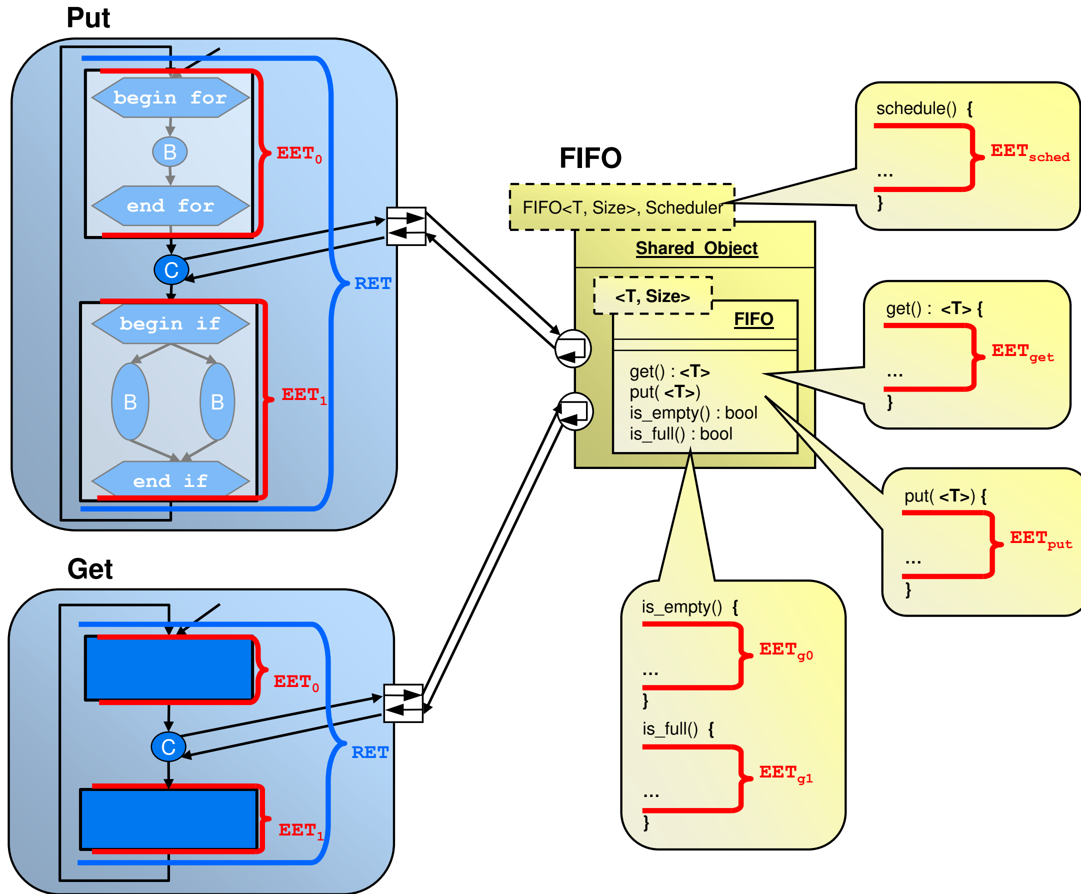


Figure 5.44: Bounded FIFO producer-consumer example

Figure 5.44 shows a simple producer-consumer Application Layer example with a bounded FIFO Shared Object. It is the same example already used in Section 5.5.4. $\mathtt{EET}_0 = [BCET, WCET]$ is transformed into a state invariant `x<=WCET` and the guard expression `x>=BCET && x<= WCET` on all outgoing transitions. Clock variable x needs to be reset $x = 0$ on all entering transitions of the state with the invariant.

5.5.6.1 Actor

Figure 5.45 models the `Put` and `Get` Actors of the example in Figure 5.44. Since both actors in the example have the same internal structure

```
1 while{true} {
2   computation of duration EET0 ⟶ represented as ''compute_before'' state
3   communication with FIFO Shared Object ⟶ represented as ''called'' state
4   computation of duration EET1 ⟶ represented as ''compute_after'' state
5 }
```

both Actors are represented by the same TA template, which in indeed not the general case. The other commited states are used for untimed synchronization as references for analysis. Both EET intervals are expressed using a combination of invariant and guarded condition, as described above. The two clocks `x` and `y` are used to express the RET. I.e. clock `x` cannot be reused and reset before entering `compute_after`, because during analysis we want to check the value of x in state `done`, representing the duration of $\mathtt{RET} = \mathtt{EET}_0 + T_{called} + \mathtt{EET}_1$. T_{called} is the duration of the communication with the FIFO Shared Object.

A service call is modeled through urgent synchronization with a `Port` TA. The communication is initiated through emission of the urgent `call` event using the shared variables `call_cid` and

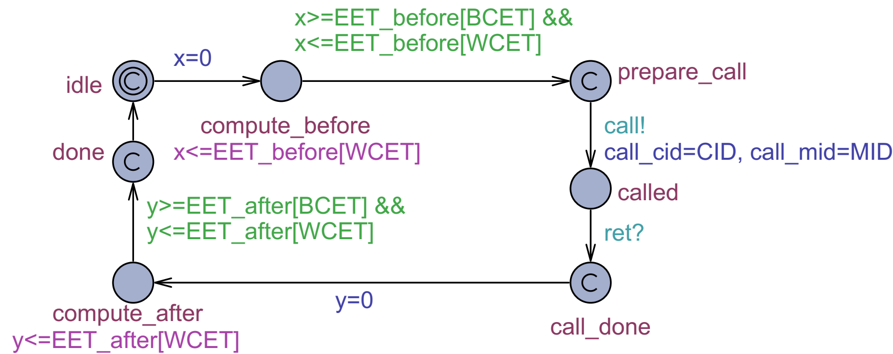


Figure 5.45: Actor modeled as timed automaton

`call_mid` to set the client ID and method ID (PUT or GET). Afterward the `called` state waits for notification of the call completion through the `ret` event.

```
1  // definition of timing annotation intervals
2  typedef int [0,1] EET_type;
3  const EET_type BCET ← 0; // lower bound: Best−Case Execution Time (BCET)
4  const EET_type WCET ← 1; // upper bound: Worst−Case Execution Time (WCET)
5
6  // number of client processes
7  const int NC ← 2;
8  typedef int [0,NC−1] client_type;
9
10 // definition of method ID types
11 typedef int [0,2] method_type;
12 const method_type NOP ← 0; // No OPeration (NOP) represents no service call
13 const method_type PUT ← 1; // represents the put service
14 const method_type GET ← 2; // represents the get service
15
16 // definition of status ID types
17 typedef int [0,2] status_type;
18 const status_type WAIT ← 0;      // waiting for access to SO
19 const status_type GRANTED ← 1;   // access to SO granted
20 const status_type COMPLETED ← 2; // access to SO has been completed
21
22 // index type for Shared Object scheduler
23 typedef int[−1,NC−1] index_type;
24 index_type granted_cid ← −1; // initially no client is granted
25
26 // definition of FIFO size
27 const int FIFO_SIZE ← 5;
```

Listing 5.15: Global definitions in the consumer-producer example

Listing 5.15 shows the global definitions used in this example:

- definition of timing annotation intervals
- definition of client ID types: each Actor (called client) has a unique ID
- definition of method ID types: each service of a Shared Object (called method) has a unique ID
- definition of communication status IDs: each communication of a client with a Shared Object has a status of this type
- definition of index type for the Shared Object's scheduler: the scheduling algorithm picks an element of this index type from a list of enabled clients (i.e. associate service request's guard has been evaluated to true)

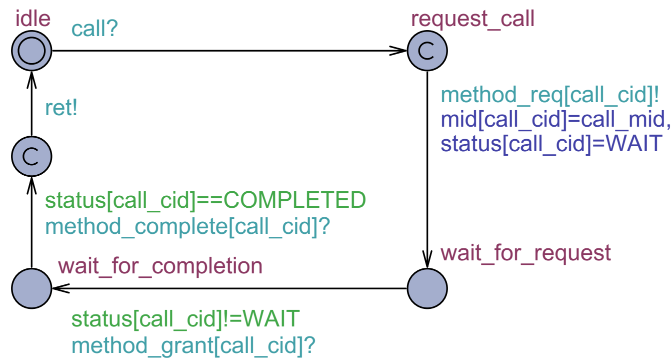


Figure 5.46: Port modeled as timed automaton

5.5.6.2 Port

The `Port` TA is a service call proxy of the `Actor` TA. Upon activation through the `call` event the service call request is registered in the Shared Object's request vector `mid[call_cid]=call_mid`, the status of the communication is set to `WAITING` and the method request event `method_req[call_cid]` is triggered. The `wait_for_request` state is left as soon as the communication status has changed to `GRANTED`. The `wait_for_completion` state is left as soon as the communication status has changed to `COMPLETED`. Afterward the `Actor` is notified about the completion of the call through the `ret` event.

On the Application Layer, the TA model of the Port does not add any timing delays to the Shared Object communication.

5.5.6.3 Shared Object

As shown in Figure 5.47, a Shared Object is split up into a *Controller*, *Arbiter*, *Guard Evaluator* and *Behavior* TA.

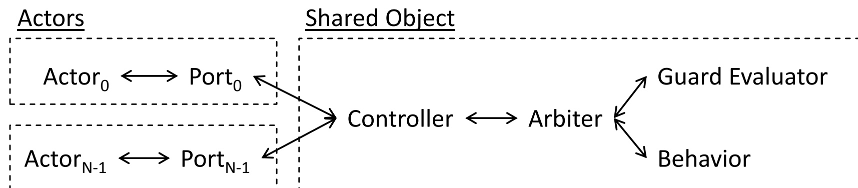


Figure 5.47: Overview of Shared Objects TA split-up

Controller The Shared Object's controller interacts with the ports of all connected Actors. From each port a dedicated broadcast channel `method_req[j]` with $j \in [0, NC-1]$ can be used to trigger the Controller. The Uppaal select statement `j : int[0,NC-1]` is used to describe sensitivity on all channels. The committed `start` state triggers the Shared Object's Arbiter TA and waits for getting granted. When access is granted, the port gets informed through the `method_grant` event. Afterward the method execution is triggered via the Arbiter TA through the `exec_method` event. After the requested method has been completed `done_method`, the Port TA gets triggered through the `method_complete` event.

On the Application Layer, the TA model of the Shared Object Controller does not add any timing delays to the Shared Object communication.

Arbiter The Shared Object Arbiter as shown in Figure 5.49 is triggered by the Controller TA through the `arbitrate` event and triggers the Guard Evaluator TA. The Guard Evaluator goes through the request vector and filters out all requested whose guards evaluate to false. This

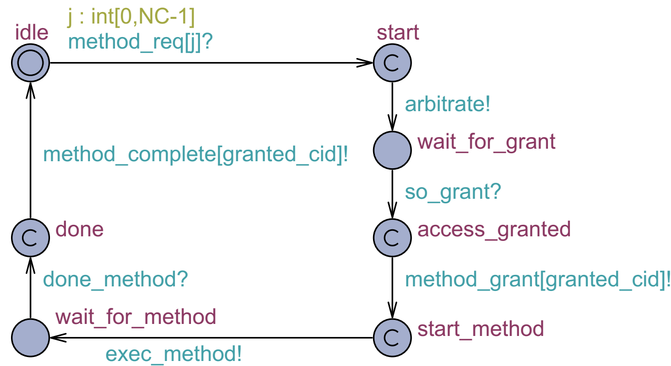


Figure 5.48: Shared Object controller modeled as timed automaton

filtering guarantees that only enabled service requests are scheduled. More details about the Guard Evaluator are described in the next paragraph.

After guard evaluation has been completed a scheduling function is called. The pre-defined scheduling functions are:

- `index_type schedule_static_prio(bool zero_is_highest)`
- `index_type schedule_ceiling_priority()`
- `index_type schedule_round_robin()`
- `index_type schedule_modified_round_robin()`
- `index_type schedule_least_recently_used()`

and can be found in Appendix B, Section B.1. Each scheduling algorithm gets a vector client IDs with enabled services requests, selects and returns the index of the scheduled client ID.

The `schedule` state is annotated with scheduling timing interval, called $\texttt{EET}_{Sched}$ as describe in Section 5.5.4.

After scheduling has been completed, the Controller TA is informed about the successful scheduling, the granted method ID is computed `mid_so=mid_guarded[granted_cid]` and the status of the granted client is updated `status[granted_cid]=GRANTED`.

After the Shared Object Controller enables the execution of the granted method, the method is executed upon `call_so` event in the Shared Object's Behavior TA (see below). After completion of the method, notified by the `ret_so` event, the Controller TA is notified about the completion of the call and the method IDs and the communication status is reset.

Guard Evaluator The Guard Evaluator, as shown in Figure 5.50, is triggered by the Arbiter TA before applying the scheduling algorithm on the request vector. The delay of the guard evaluation is modeled again as $[BCET, WCET]$ interval in the `eval` state. The `evaluate()` function shown in Listing 5.16 takes the plain request vector `req_in` of size NC (number of clients) and overwrites all service requests with `NOP` whose guard condition is not fulfilled.

```
1  bool is_full () { return num_elements == FIFO_SIZE; }
2  bool is_empty() { return num_elements == 0; }
3
4  void evaluate() {
5      for(i : int [0,NC−1]) {
6          if (req_in[i] == PUT ∧ !is_full()) req_out[i] ← PUT;
7          else if (req_in[i] == GET ∧ !is_empty()) req_out[i] ← GET;
8          else req_out[i] ← NOP;
9      }
10 }
```

Listing 5.16: Functions used in the Shared Object guard evaluator model in Figure 5.50

In the FIFO example, with the guarded methods:

```
OSSS_GUARDED_METHOD_VOID(put, OSSS_PARAMS(1, T, item), !is_full() )
OSSS_GUARDED_METHOD(T, get, OSSS_PARAMS(0), !is_empty() )
```

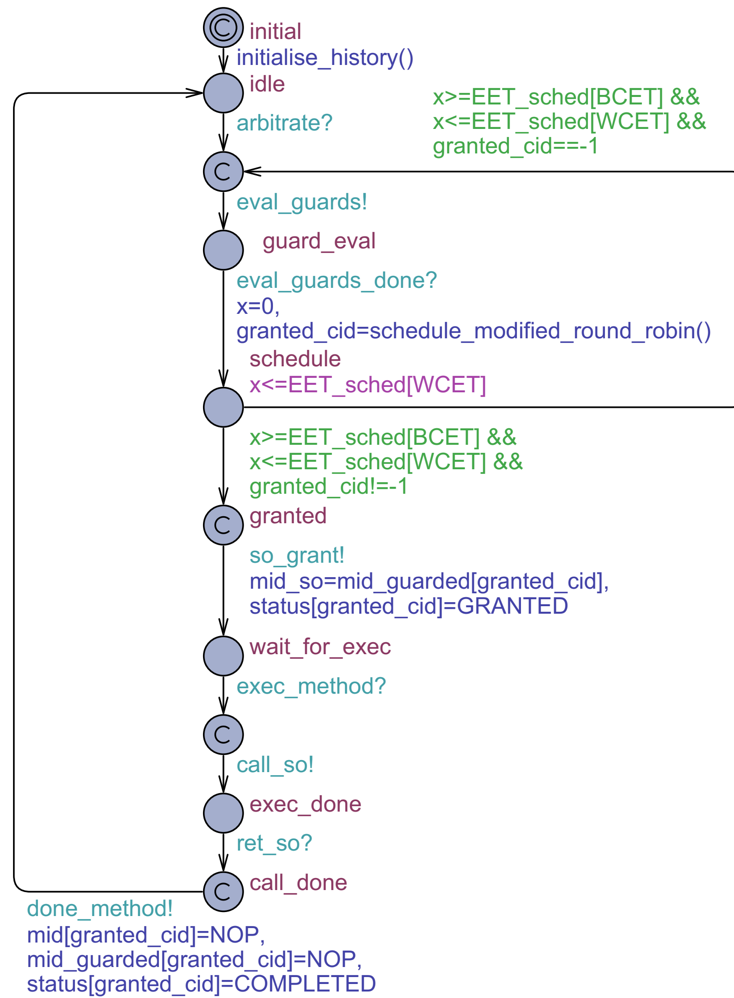


Figure 5.49: Shared Object arbiter modeled as timed automaton

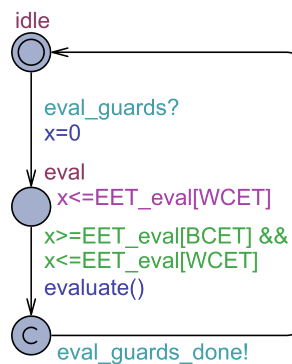


Figure 5.50: Shared Object guard evaluator modeled as timed automaton

the following conditions are evaluated:

```
if (req_in[i] == PUT && !is_full()) then req_out[i] = PUT
if (req_in[i] == GET && !is_empty()) then req_out[i] = GET
```

Otherwise, the guard condition is not fulfilled and the requested service call is masked

```
req_out[i] = NOP
```

Behavior The provided services of the Shared Object are modeled in the Behavior TA (see Figure 5.51) which is triggered be the Arbiter TA after guard evaluation and scheduling. Since a Shared Object is passive and cannot call services of another Shared Object, the resulting TA represents a look-up table, with a dedicated state per service. The delay of each service is modeled as $[BCET, WCET]$ interval.

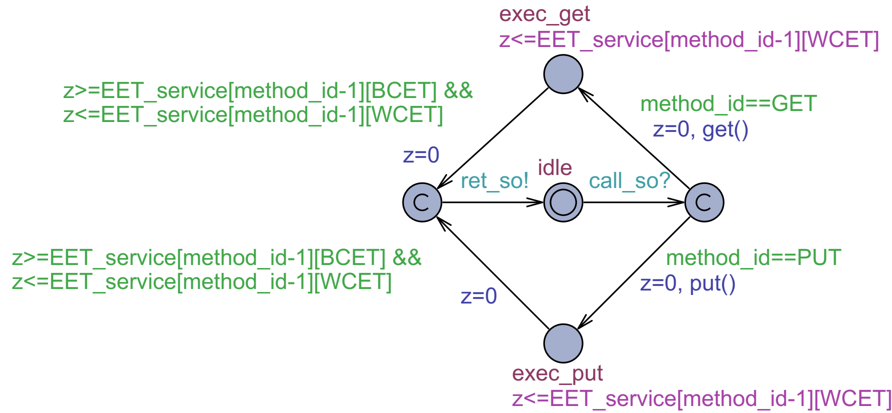


Figure 5.51: Shared Object behavior modeled as timed automaton

Side-effects of the service calls that change the local state of the Shared Object and have an impact on the guard conditions need to be modeled explicitly. For the FIFO Shared Object the side-effects on the total number of FIFO elements (`num_elements`) cause by the `put` and `get` services are modeled through the functions shown in Listing 5.17.

```
1  void put() { num_elements ← num_elements + 1; }
2  void get() { num_elements ← num_elements − 1; }
```

Listing 5.17: Functions used in the Shared Object behavior model in Figure 5.51

5.5.6.4 Putting it all together

Listing B.3 instantiates the described TA templates to the bounded FIFO producer-consumer example shown in Figure 5.44. In line 2-12 of Listing 5.18 the following EET intervals are defined:

$$\begin{aligned}
Put.EET_0 &:= \texttt{put_EET_before} = [10, 15] \\
Put.EET_1 &:= \texttt{put_EET_after} = [4, 5] \\
Get.EET_0 &:= \texttt{get_EET_before} = [10, 15] \\
Get.EET_1 &:= \texttt{get_EET_after} = [4, 5] \\
FIFO.T_{GE} &:= \texttt{EET_eval} = [1, 1] \leq NC \cdot \max\left(FIFO.EET_{g0}, FIFO.EET_{g1}\right) \\
FIFO.EET_{sched} &:= \texttt{EET_sched} = [1, 1] \\
FIFO.EET_{put} &:= \texttt{EET_service[0]} = [2, 5] \\
FIFO.EET_{get} &:= \texttt{EET_service[1]} = [2, 5]
\end{aligned}$$

RET upper bounds are defined in line 15+16:

$$Put.RET := \texttt{PUT_PERIOD} = 55$$
$$Get.RET := \texttt{GET_PERIOD} = 55$$

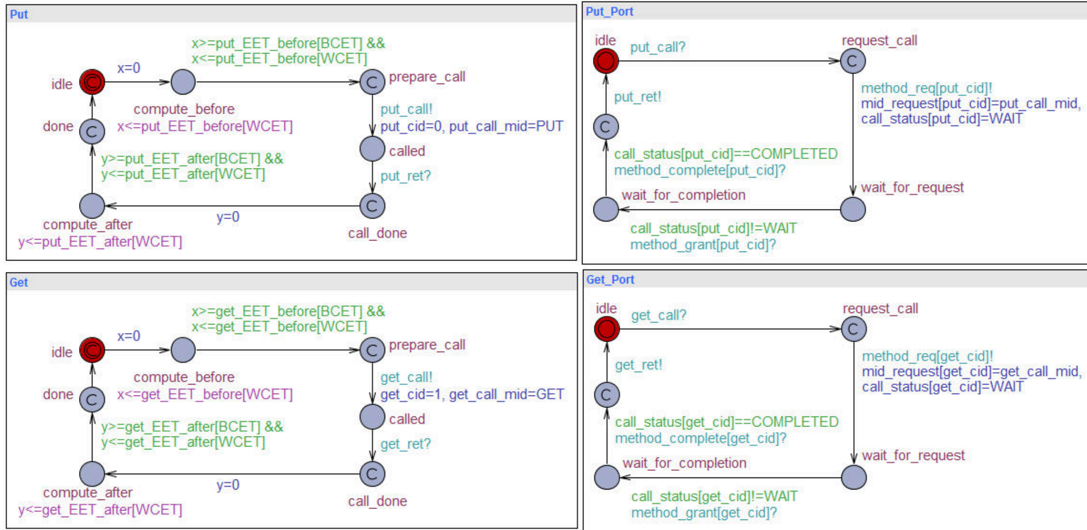


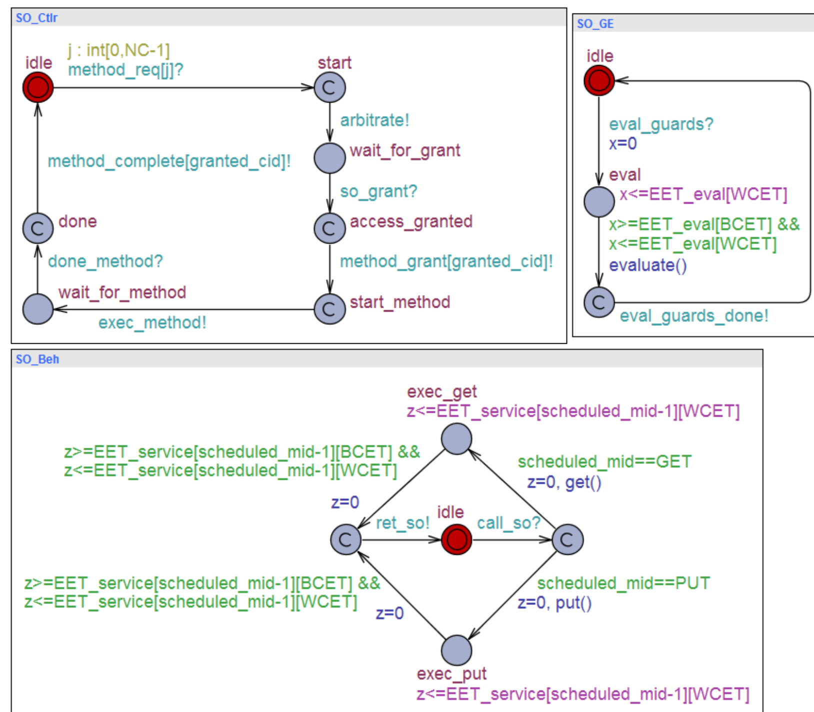
Figure 5.52: Consumer Producer System: Actors with Ports

Figure 5.52 shows the instantiation of the `Put` and `Get` Actors with their associated ports `Put_Port` and `Get_Port`. Figure 5.53a shows the FIFO Shared Object's Controller `SO_Ctlr`, Guard Evaluator `SO_GE` and Behavior `SO_Beh`. Figure 5.53b shows the Shared Obejct's Arbiter `SO_Arb`.

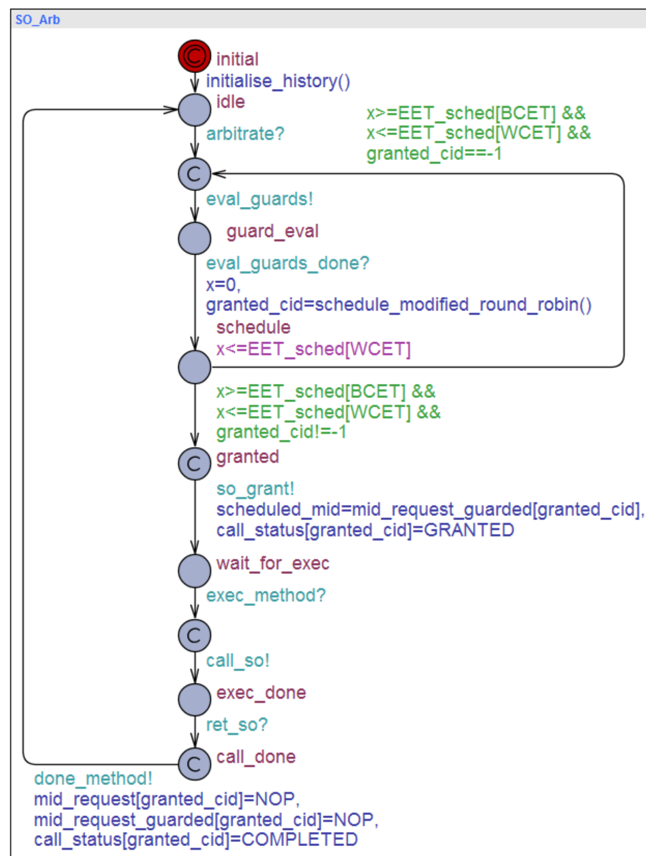
```
1  // Timing annotations, written as [BCET, WCET] intervals
2  int put_EET_before[2] ← {10, 15};
3  int put_EET_after[2]  ← {4, 5};
4
5  int get_EET_before[2] ← {10, 15};
6  int get_EET_after[2]  ← {4, 5};
7
8  int EET_eval[2] ← {1, 1};
9  int EET_sched[2] ← {1, 1};
10
11 int[0,FIFO_SIZE] num_elements ← 0;
12 int EET_service[2][2] ← {{2, 5}, {2, 5}};
13
14 // timing requirements for put and get clients
15 const int PUT_PERIOD ← 55;
16 const int GET_PERIOD ← 55;
```

Listing 5.18: Timing annotations of the consumer-producer example

```
1  method_type mid_request[NC];
2  method_type mid_request_guarded[NC];
3  method_type so_mid ← NOP;
4  status_type call_status[NC];
5
6  urgent chan put_call, put_ret;
7  client_type put_cid;
8  method_type put_call_mid;
9
10 Put ← Actor(0, PUT, put_cid, put_call_mid, put_call, put_ret,
11             put_EET_before, put_EET_after);
12
13 // to Shared Object Controller
14 broadcast chan method_req[NC];
```



(a) Controller, Guard Evaluator and Behavior



(b) Arbiter

Figure 5.53: Consumer Producer System: Shared Object

```
15 chan method_grant[NC];
16 // from Shared Object Controller
17 urgent chan method_complete[NC];
18
19 Put_Port ← Port(put_cid, put_call_mid, put_call, put_ret,
20                 method_req, method_grant,
21                 mid_request, call_status,
22                 method_complete);
23
24 urgent chan get_call, get_ret;
25 client_type get_cid;
26 method_type get_call_mid;
27
28 Get ← Actor(1, GET, get_cid, get_call_mid, get_call, get_ret,
29             get_EET_before, get_EET_after);
30
31 Get_Port ← Port(get_cid, get_call_mid, get_call, get_ret,
32                 method_req, method_grant,
33                 mid_request, call_status,
34                 method_complete);
35
36 // to Arbiter
37 urgent broadcast chan arbitrate;
38 urgent chan so_grant, exec_method, done_method;
39 method_type scheduled_mid;
40
41 SO_Ctlr ← SO_Controller(method_req,
42                         arbitrate, so_grant, granted_cid,
43                         method_grant,
44                         exec_method, done_method,
45                         method_complete);
46
47 // Arbiter to Server
48 urgent chan call_so, ret_so;
49 // Arbiter to Guard Evaluator
50 urgent chan eval_guards, eval_guards_done;
51
52 SO_GE ← SO_Guard_Evaluator(eval_guards, eval_guards_done,
53                            mid_request, mid_request_guarded,
54                            num_elements,
55                            EET_eval);
56
57 SO_Arb ← SO_Arbiter(arbitrate,
58                     eval_guards, eval_guards_done,
59                     so_grant,
60                     mid_request, mid_request_guarded, call_status,
61                     exec_method, done_method,
62                     call_so, granted_cid, scheduled_mid, ret_so,
63                     EET_sched);
64
65 SO_Beh ← SO_Behavior(call_so, ret_so, scheduled_mid,
66                      EET_service, num_elements);
67
68 system Put, Put_Port,
69        Get, Get_Port,
70        SO_Ctlr, SO_GE, SO_Arb, SO_Beh;
```

Listing 5.19: System definition of the consumer-producer example

5.5.6.5 Properties

The main purpose of a formal model is to verify it with respect to a requirement specification. Like the model, the requirement specification needs to be expressed in a formally well-defined language. Uppaal uses a simplified version of TCTL (Timed Computation Tree Logic). Like in TCTL, the query language consists of path formulae and state formulae. State formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be classified into reachability, safety and liveness. More information on the used query language can be found in Appendix 3.3.4.

Safety properties are of the form: “something bad will never happen”. A variation of this property is that “something will possibly never happen”.

In Uppaal these properties are formulated positively, e.g., something good is invariantly true. Let φ be a state formulae. The path formulae $A\Box\varphi$ expresses that φ should be true in all reachable states. While $E\Box\varphi$ says that there should exist a maximal path[1] such that φ is always true.

For the FIFO Shared Object example system in Listing B.3 the following safety properties are fulfilled:

- $A\Box$ (*not deadlock*)

  The system does never deadlock.
- $A\Box$ (`Put.done` $\Rightarrow$ `Put.x` $\leq$ `PUT_PERIOD`)

  The completion of a `put` call takes never longer than `PUT_PERIOD`.
- $A\Box$ (`Get.done` $\Rightarrow$ `Get.x` $\leq$ `GET_PERIOD`)

  The completion of a `get` call takes never longer than `GET_PERIOD`
- $A\Box$ ((`Put.call_done` $\Rightarrow$ *not* `Get.call_done`) *and*
  (`Get.call_done` $\Rightarrow$ *not* `Put.call_done`))

  The `put` and `get` service call executions are always mutual exclusive.
- $A\Box$ (`num_elements` $\leq$ `FIFO_LIMIT`)

  The total number of elements in the Shared Object’s buffer is always less or equal to `FIFO_LIMIT`. Given the timing annotation from Listing 5.18 `FIFO_LIMIT` depends on the following parameters:

priority	scheduling algorithm	FIFO_LIMIT
Put > Get	static priority	FIFO_SIZE:=5
Put < Get	static priority	2
-	ceiling priority	2
-	round robin	2
-	modified round robin	2

Liveness properties are of the form: “something will eventually happen”. In its simple form, liveness is expressed with the path formula $A\Diamond\varphi$, meaning φ is eventually satisfied. The more useful form is the “leads to” or “response” property, written $\varphi \leadsto \psi$ which is read as whenever φ is satisfied, then eventually ψ will be satisfied, e.g. whenever a message is sent, then eventually it will be received. $\varphi \leadsto \psi$ is equivalent to $A\Box(\varphi \Rightarrow A\Diamond\psi)$.

For the FIFO Shared Object example system in Listing B.3 the following liveness properties are fulfilled:

- `Get.called` $\leadsto$ `Get.call_done`

  When a `get` call is requested it will be eventually served by the Shared Object.
- `Put.called` $\leadsto$ `Put.call_done`

  When a `put` call is requested it will be eventually served by the Shared Object.
- `Put.call_done` $\leadsto$ `Get.call_done`

  When a `put` call is completed a `get` call eventually completes.

[1] A maximal path is a path that is either infinite or where the last state has no outgoing transitions.

5.6 Virtual Target Architecture Layer

5.6.1 Introduction

The *Application Layer Model* must be further refined and mapped to a virtual architecture to determine the implementation of the functional model and to enable the synthesis process. In order to separate the description of the pure application from the architecture it is mapped to, we provide a third layer called *Virtual Target Architecture Layer*. On this layer several architecture building blocks are available which can be used to assemble the overall system architecture. These building blocks are software processors, memories and (user defined) hardware blocks. For the interconnection of these blocks different communication networks, like buses or high speed point-to-point connections are available.

The mapping step from the *Application Layer* to the *Virtual Target Architecture Layer* involves the mapping of Actors and Shared Objects to appropriate architecture building blocks. Actors which should be implemented in software have to be mapped on a software processor, while Actors and Shared Objects which should be implemented in hardware have to be mapped on certain hardware blocks.

Besides the mapping of Actors and Shared Objects, the communication links defined on the *Application Layer* have to be mapped on communication resources of the *Virtual Target Architecture Layer*. Multiple communication links can be mapped to a shared communication resource (e.g. a shared bus) but also to dedicated point-to-point connections.

The main properties of the Virtual Target Architecture Layer can be summarized as:

- Main focus on execution platform architecture and configuration, last structural model before platform and custom hardware/software synthesis
- Enables integration of IP components
- Models the platform's communication infrastructure independent from the functional/behavioral part (i.e. refinement of the communication links between Actors and Shared Objects from the Application Layer)
- Adds communication behavior and communication time to the Application Layer model
- Not executable on its own, but executable after mapping Application Layer modeling elements to the provided Virtual Target Architecture modeling elements

5.6.2 Modeling Elements

The Virtual Target Architecture Layer is a structural executable[2] parallel object-oriented model. All Architecture Objects have a fixed location (i.e. inside the SoC architecture) and cannot be copied or passed by value. Architecture Objects cannot be hierarchically composed. The Virtual Target Architecture Layer consists of the following modeling elements:

- **Software Socket** is a pure structural element (does not model any behavior/functionality). Software Sockets are abstract software processors and mainly represent the communication interfaces of the processor. A Software Socket has a clock and a reset port. For communication it has a single RMI Master Port that is connected to an RMI Channel (see Definition 5.6.2.13). A Software Socket is a container for an Actor of type *reactive sequential* or *active sequential*. All communication of this Actor with Shared Objects is performed through the single RMI Master Port. Software Sockets belong to a System on Chip (see Definition 5.6.2.19).
- **Hardware Socket** is a pure structural element (does not model any behavior/functionality). Hardware Sockets are custom hardware processors and mainly represent the communication interfaces of this processor. All Hardware Sockets have a clock and a reset

[2] Some of the Virtual Target Architecture Layer elements (e.g. Software and Hardware Socket) are not executable by themselves, but after mapping of executable Application Layer elements on them, they become executable transitionally.

port and belong to a System on Chip (see Definition 5.6.2.19). This socket can be of the following two kinds:

- **Shared Object Socket** is a container for a single Shared Object. Depending on the mapping of the Shared Object's client's a Shared Object Socket has a scalable number of RMI Slave Ports (for more details see Definition 5.6.2.11). All service requests on the contained Shared Objects are performed through these RMI Slave Ports.
- **Actor Socket** is a container for an Actor of all types. Actor Sockets have a scalable number of RMI Master Ports. The number of ports depends on the kind of the contained Actor:
  - *reactive* and *active sequential*: single RMI Master Port
  - *reactive* and *active parallel*: $par_set(B)$ RMI Master Ports, where B is the Behavior inside the Socket

  All communication of the Actor (inside an Actor Socket) with Shared Objects is performed through the Actor Socket's RMI Master Ports.

- **Memory** elements model a continuous physical area of memory. It might represent an off-chip memory or an internal SRAM or FPGA Block-RAM (BRAM). Memories can be used inside the Behavior(s) of Actors and Shared Objects. A Memory block has a dedicated static total size in bits. Basic data types and passive objects can be mapped into a Memory Element. Memory elements cannot be accessed directly, but through accessor proxies. This proxy defines the read and write access granularity (in bits) and timing penalties/delays. For multi-ported memories each port is modeled by a dedicated accessor. All objects to be stored inside a Memory element require to be serializable (see Definition 5.6.2.4).

- **Remote Method Invocation (RMI) Channel** is the communication medium on the Virtual Target Architecture for service calls of Actors to Shared Objects. An RMI Channel has a clock and a reset port. It implements an RMI Master Interface for the connection with RMI Master Ports and and RMI Slave Interface for the connection with RMI Slave Ports. The RMI Channel is a hierarchical channel. It contains either a Bus Channel or a Point-to-Point Channel. The RMI Channel performs the RMI protocol, argument and return value serialization and de-serialization using the basic single beat or burst transfer services of the Bus or Point-to-Point Channel. In this work the following generic Channels are provided:
  - **Simple Bus** has a configurable data and address bus bit-width. The optional bus arbiter[3] can use the same scheduling algorithms as Shared Objects (see Section 5.5.3). The Simple Bus supports single beat (transmits a single data chunk of the configured data bus bit-width) and burst (transmits a dynamically configurable amount of data chunks of the configured bus bit-with, one per clock-cycle without interruption) transfers. The simple bus also takes care of the address handling. Each bus slave/target port has an associated address range (basic address + size in granularity of data bus bit-width). The bus maps the targeted address at the master/initiator port into the activation of the right slave/target port.
  - **Simple Point-to-Point (P2P) Channel** has a single master/initiator and a single slave/target. The channel is bidirectional with independently configurable send and return bit-width.

- **IP Components** are pre-existing synthesizable hardware elements at RT-level. Communication with IP components is established via signals. IP Components can be connected with hardware sockets only.

- **System on Chip** is a pure structural element (does not model any behavior/functionality) and represents the structural boundary of the modeled SoC. It has a clock and reset port

[3] The bus arbiter is only used with multiple bus masters/initiators.

and other ports for off-chip communication. A System on Chip element can contain Software Sockets, Hardware Sockets and RMI Channels.

Figure 5.54 gives an overview of the Virtual Target Architecture Meta Model. In the following subsections these modeling elements will be defined.

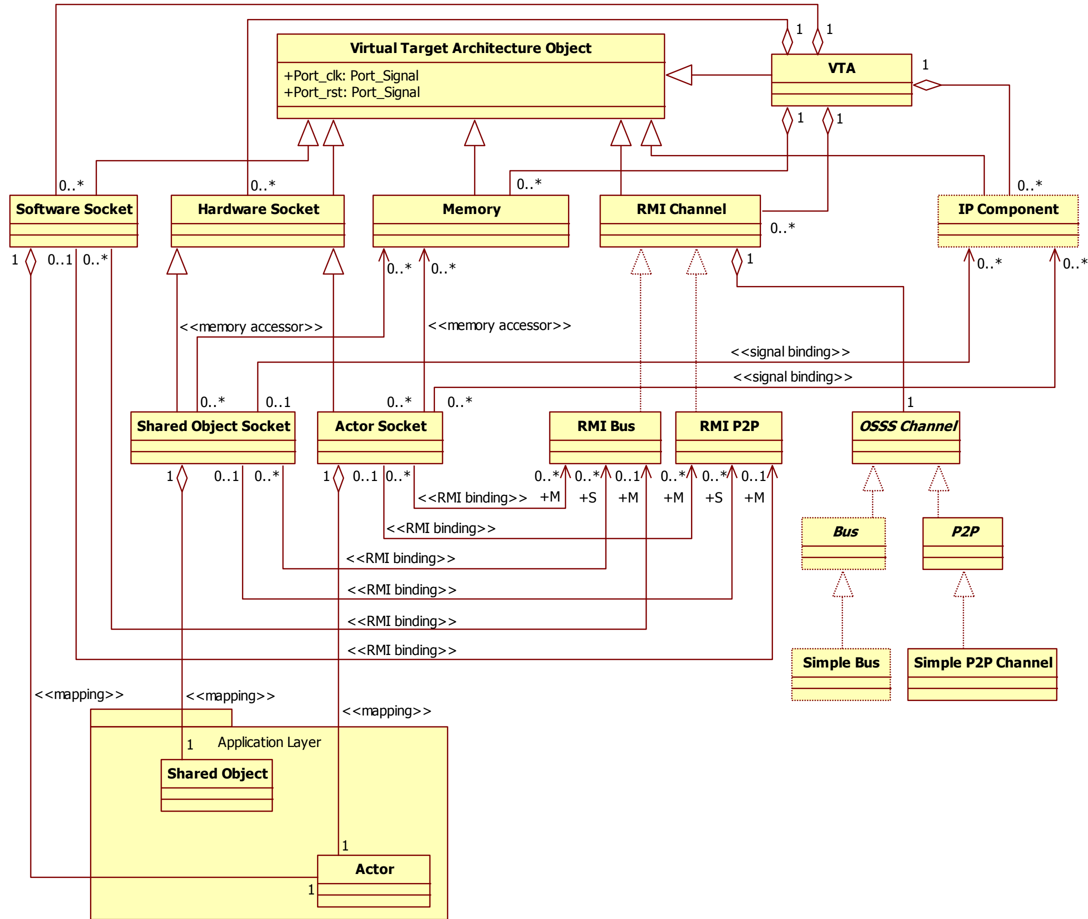


Figure 5.54: Virtual Target Architecture Meta Model

5.6.2.1 Signal and Signal Port

Definition 5.6.2.1 (Signal):
A Signal, called Signal = [DataType] is a simple channel used to exchange plain data of basic type (see Definition 5.3.1.1) or arrays of basic type (see Definition 5.3.3.1). A signal provides interfaces for writing, reading of the following kinds:

1. signal_in_if<Data Type> *provides write access to the signal*

$$write\colon DataType \to void$$

2. signal_out_if<Data Type> *provides read access to the signal*

$$read\colon void \to DataType$$

3. signal_inout_if<Data Type> *provides read and write access to the signal*

$$write\colon DataType \to void$$
$$read\colon void \to DataType$$

□

Definition 5.6.2.2 (Signal Port):
A Signal Port, called $Port_{Signal} = [DataType, Required_{Signal_{IF}}, Bound_{Signal_{IF}}]$*, is a special Port (see Definition 5.4.2.1) where DataType can be any basic type or an array of any basic type. Signal Ports can be of the following kinds:*

- *Signal In Port:* $Port_{Signal_in}$ *with*
  - $Required_{Signal_{IF}} := signal_in_IF < DataType >$ *and*
  - $Bound_{Signal_{IF}} \in \mathbf{Signal} < \mathbf{DataType} >$
- *Signal Out Port:* $Port_{Signal_out}$ *with*
  - $Required_{Signal_{IF}} := signal_out_IF < DataType >$ *and*
  - $Bound_{Signal_{IF}} \in \mathbf{Signal} < \mathbf{DataType} >$
- *Signal In-Out Port:* $Port_{Signal_inout}$ *with*
  - $Required_{Signal_{IF}} := signal_inout_IF < DataType >$ *and*
  - $Bound_{Signal_{IF}} \in \mathbf{Signal} < \mathbf{DataType} >$

□

5.6.2.2 RMI Port

Definition 5.6.2.3 (RMI Port):
A Remote Method Invocation (RMI) Port, called $Port_{RMI} = [Required_{RMI_{IF}}, Bound_{RMI_{IF}}]$*, is a special Port (see Definition 5.4.2.1) of the following kinds:*

- *RMI Client Port:* $Port_{RMI_C}$ *with*
  - $Required_{RMI_{IF}} := RMI_Client_IF$ *and*
  - $Bound_{RMI_{IF}} \in$ **RMI Channel**
- *RMI Server Port:* $Port_{RMI_S}$ *with*
  - $Required_{RMI_{IF}} := RMI_Server_IF$ *and*
  - $Bound_{RMI_{IF}} \in$ **RMI Channel**

Where RMI_Client_IF is the RMI client interface, RMI_Server_IF is the RMI server interface and **RMI Channel** *is the set of all RMI Channels. Mode details on the RMI Channel and its interfaces can be found in Definition 5.6.2.13.* □

5.6.2.3 Serializable Object

In general, *Serialization* is the conversion of an object to a series of bits or bytes, so that the object can be easily saved to persistent storage or streamed across a communication link. The bit or byte stream can then be deserialised and converted into a replica of the original object.

Definition 5.6.2.4 (Serializable Object):
A Serializable Object is defined by a ***Serializable Class****, which is a tuple*

$$\overleftarrow{C} = [c_{parent} \cup serializable_object, State, Method]$$

with:

1. *A single user-defined base class* $c_{parent} \in \mathcal{C}$ *(single inheritance)*
2. *and the* ***serializable_object*** *base class.*

3. *A state vector* $State = T_0 \times \cdots \times T_n$, *with* $n \geq 0$ *where* $T_0, \ldots T_n$ *denote some basic types, classes, or arrays. The state vector consists of a volatile (part of the state space not to be serialized and deserialized) and a non-volatile (part of the state space to be serialized and deserialized) partition with the following property:* $State = VState \cup NVState$ *and* $VState \cap NVState = \emptyset$. *When restoring an object where* $VState \neq \emptyset$, *all elements of this volatile sub-state-space will be initialized with the default values during object construction. All elements of the non-volatile sub-state-space will retain their values.*
4. *A set of member functions* $Method = \{m_0, \ldots m_m\}$ *with* m_i, $i \geq 0$ *of the following kinds:*

$name\colon \textbf{\textit{void}} \rightarrow \textbf{\textit{void}}$	*no arguments and no return type*
$name\colon \textbf{\textit{void}} \rightarrow T$	*no arguments and return type* T
$name\colon T_0 \times \cdots \times T_n \rightarrow \textbf{\textit{void}}$	*arguments* T_i, $i \geq 0$ *and no return type*
$name\colon T_0 \times \cdots \times T_n \rightarrow T$	*arguments* T_i, $i \geq 0$ *and return type* T

Including the two mandatory serialization and deserialization methods:

$serialize\colon \textbf{\textit{State}} \rightarrow \textbf{\textit{NVState}}$	*maps the state of the class to a non-volatile sub-state, where* $\textbf{\textit{NVState}} \subseteq \textbf{\textit{State}}$
$deserialize\colon \textbf{\textit{NVState}} \rightarrow \textbf{\textit{State}}$	*maps the non-volatile sub-state of the class into the overall state, where* $\textbf{\textit{NVState}} \subseteq \textbf{\textit{State}}$

A Serializable Class needs a ***Default Constructor*** *and a* ***Copy Constructor****.* □

The representation of the object as a bit or byte stream is implementation dependent. In the following, we assume that the bit or byte stream representation is dense (i.e. the total size of the bit stream is sum of the bit sizes of all members) and no padding is used.

Serialization support for the Basic Types *Integer* and *Boolean*, as well as *Arrays* of Integer, Boolean, Bit or user-defined classes is provided.

Classes that contain data members of non-serializable classes in their state vectors are not serializable as a whole, until all members and members of members, etc. are serializable.

5.6.2.4 Virtual Target Architecture Object

Definition 5.6.2.5 (Virtual Target Architecture Object (VTAO)):
The Virtual Target Architecture Object is the base-class of all Virtual Target Architecture Layer objects. It only consists of a module with clock and reset port. It is defined as $VTAO = [Port_{clk}, Port_{rst}]$, *where*

- $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*
- $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) clock synchronous reset port.*

All Virtual Target Architecture Layer objects inherit a clock and a reset port from the VTAO. □

5.6.2.5 Memory

Definition 5.6.2.6 (Memory):
A Memory is a tuple $Memory = [Port_{clk}, Port_{rst}, Size, \overline{Accessor}, Type]$, *where*

- $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*
- $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) reset port.*
- $Size \in \mathbb{N}_{>0}$ *is the total size or capacity of the memory in Bits, with* $Size \bmod 8 = 0$ *(only multiples of 8 Bit allowed).*

- $\overline{Accessor}$ *is a vector of* $MemoryAccessor$ *elements as defined in Definition 5.6.2.7. When more than one Accessor* $|\overline{Accessor}| > 1$ *is connected the a memory, no guarantee for data integrity can be provided. I.e. concurrent accesses to the memory are not arbitrated*[4].
- $Type \in \{None, Basic\ Type, Object, Basic\ Type\ Array, Object\ Array\}$ *is the data type (see Section 5.3) mapped to this memory.*

□

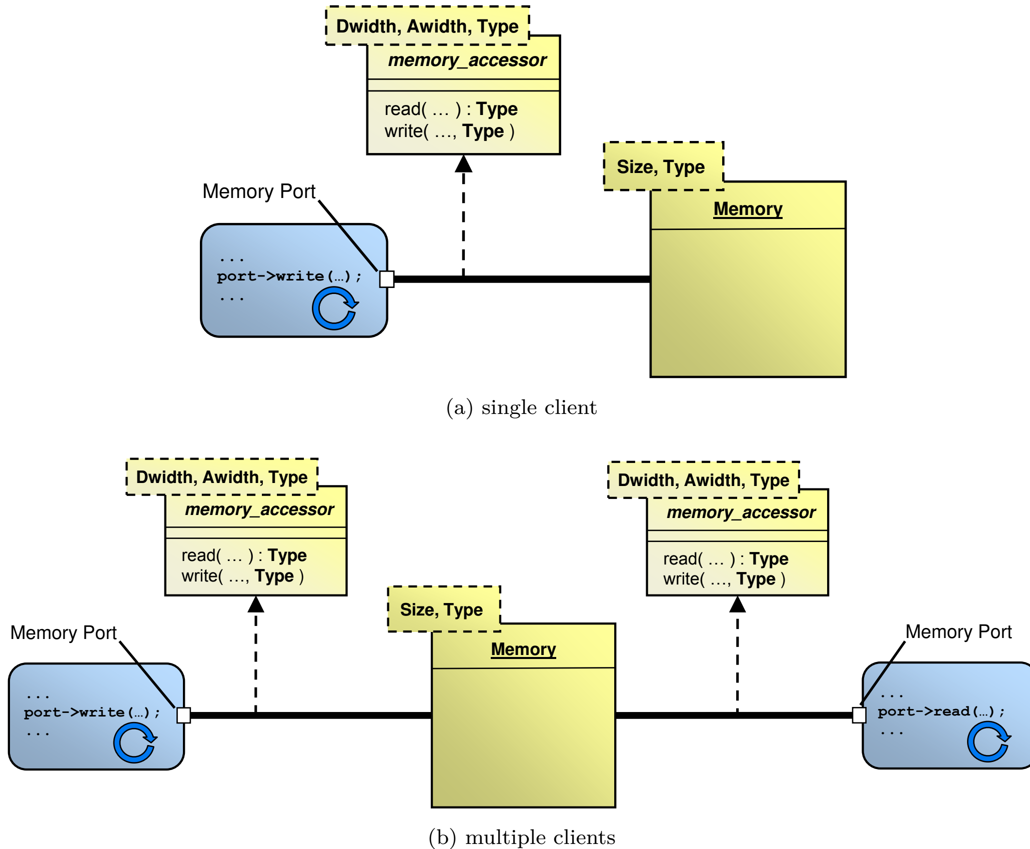


Figure 5.55: Memory and Memory Accessor

Figure 5.55 depicts the usage of the *Memory* and *Memory Accessor* elements. The *Memory* memory element represents a physical storage container of a specific size (in bit). A *Memory Accessor* is a channel used to connect hardware clients (HW Actors or Shared Objects) with the Memory. The Accessor restricts the access granularity to a certain data and address width and qualifies the logical addressing of the memory content (raging from raw bit-level to object array access). Figure 5.55a depicts the connection of a Hardware Actor with a Memory using a Memory Accessor Channel as communication link. In Figure 5.55b two Hardware Actors are accessing the same Memory element. This kind of shared memory access requires a memory element with multiple ports (e.g. a dual-ported memory). No guarantee regarding access arbitration is provided. Using a Shared Object for arbitrating access to a shared memory instead, allows deterministic access scheduling and should be preferred in most cases.

Definition 5.6.2.7 (Memory Accessor):
A Memory Accessor is a Channel defined as a tuple

$$MemoryAccessor = [M, DWidth, AWidth, BoundPort],\ where$$

- $M \in$ **Memory** *is the Memory (see Definition 5.6.2.6) the Accessor is bound to.*

[4] When arbitrated memory access is required, a combination of a Shared Object Wrapper (see Definition 5.6.2.11) in combination with a *Memory* is recommended.

- $DWidth, AWidth \in \mathbb{N}_{>0}$ *is the data and address width of the Accessor in Bits. In order to match with the total size of the memory* M *the following constraints for* $DWidth$ *and* $AWidth$ *apply:*

$$M.Size \geq DWidth \cdot 2^{AWidth}$$

- $BoundPort \in \mathbf{Port_{Mem}}$ *is the Memory Port (see Definition 5.6.2.8) that is bound to this memory accessor channel. The configuration of the memory accessor depends on the interface type of the Memory Port. The configuration consists of:*

  1. $mode \in \{read, write, read/write\}$ *is the access mode.*
  2. $kind \in \{raw, simple, array\}$ *is the kind of access granularity with the following meaning:*

     raw: *defines a raw bit-level access to the memory. The granularity of this raw bit-level access is defined by data width (DWidth) and address width (AWidth) parameters.*

     simple: *defines a data type aligned access to the memory. It enables access to the entire value of a basic type or to the public members of a class stored in the memory. It also enables to access a complete array of basic types or objects.*

     array: *defines an array and data type aligned access to the memory. For an array of a basic type or an array of a class type stored in the memory, it enables access to any element of the respective array (within the bounds of the array). Additionally it allows the same data type aligned access (like simple) within each array element.*

     The supported set of access granularities depends on M.Type in the following way:

     (a) ***if*** $M.Type == None$ ***then*** $kind \in \{raw\}$

     (b) ***if*** $M.Type == Basic\ Type$ ***then*** $kind \in \{raw, simple\}$

     (c) ***if*** $M.Type == Object$ ***then*** $kind \in \{raw, simple\}$

     (d) ***if*** $M.Type == Basic\ Type\ Array$ ***then*** $kind \in \{raw, simple, array\}$

     (e) ***if*** $M.Type == Object\ Array$ ***then*** $kind \in \{raw, simple, array\}$

To support the services of all possible configurations, the Memory Accessor Channel implements all interfaces shown in Table 5.3. These interfaces provide the following services (only combined read/write interfaces shown, read interfaces only provide the read services and write interfaces only provide the write services):

memory_read_write_raw_if $\langle$ ***AWidth, DWidth*** $\rangle$ *Provides raw data access. Address range is within the interval* $0, \ldots, 2^{AWidth} - 1$. *Data is a Bitvector of size* $DWidth$.

$$\begin{aligned} read &: [0, 2^{AWidth} - 1] \rightarrow Bitvector[DWidth] \\ write &: [0, 2^{AWidth} - 1] \times Bitvector[DWidth] \rightarrow void \end{aligned}$$

memory_read_write_simple_if $\langle$ ***T*** $\rangle$ *Provides access to a data type* T. *If this data type* T *is a user defined class, this interface offers direct access to the all public members of this class.* $public_member\langle T\rangle(m)$ *is a selector for any public member* m *of class* T. $type_of\langle public_member\langle T\rangle(m)\rangle$ *determines the type of the public member* m *of class* T. *If a selected member is a again a user defined class with public members, the* $access_helper\langle\cdots\rangle$ *function can be used to chain several read or write services (e.g.* $read(read(m_1), m_2)$ *reads member* m_2 *of member* m_1 *and* $write(write(m_1), m_2, value)$ *write value to member* m_2 *of member* m_1*).*

$$\begin{aligned} read &: void \rightarrow T \\ read &: public_member\langle T\rangle(m) \rightarrow type_of\langle public_member\langle T\rangle(m)\rangle \\ read &: public_member\langle T\rangle(m) \rightarrow access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle \\ read &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow \end{aligned}$$

$$
\begin{aligned}
& \quad type_of\langle public_member\langle T\rangle(m)\rangle \\
read &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow \\
& \quad access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
write &: T \rightarrow void \\
write &: public_member\langle T\rangle(m) \times type_of\langle public_member\langle T\rangle(m)\rangle \rightarrow void \\
write &: public_member\langle T\rangle(m) \rightarrow access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle \\
write &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \times \\
& \quad type_of\langle public_member\langle T\rangle(m)\rangle \rightarrow void \\
write &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow \\
& \quad access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle
\end{aligned}
$$

***memory_read_write_array_if* ⟨ *T* ⟩** *Provides the same services as the memory_read_write_simple_if*$\langle T\rangle$ *with additional access to array elements. If* T *is an array with num_elements*$\langle T\rangle$ *of type* $T[]$ *the index function* $read(i)$ *returns the* i*-th element with* $i \in [0, num_elements\langle T\rangle - 1]$. *The index function* $write(i, value)$ *write value to the* i*-th element respectively.*
For multi-dimensional arrays the read and write index functions can be chained, e.g. $read(read(read(i), j)k)$ *to read the* k*-th of the* j*-th of the* i*-th element of a three dimensional array,* $write(write(write(i), j)k, value)$ *writes value to the* k*-th of the* j*-th of the* i*-th element in the same way.*
Array access can also be arbitrarily mixed with direct or chained public member access of user-defined classes, e.g. $read(read(m_1), i)$ *reads array element* i *of member* m_1 *and* $write(write(m_1), i, value)$ *write value to array position* i *of member* m_1.

$$
\begin{aligned}
read &: void \rightarrow T \\
read &: public_member\langle T\rangle(m) \rightarrow type_of\langle public_member\langle T\rangle(m)\rangle \\
read &: public_member\langle T\rangle(m) \rightarrow access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle \\
read &: [0, num_elements\langle T\rangle - 1] \rightarrow T[\,] \\
read &: access_helper\langle T\rangle \times [0, num_elements\langle T\rangle - 1] \rightarrow T[\,] \\
read &: access_helper\langle T\rangle \times [0, num_elements\langle T\rangle - 1] \rightarrow access_helper\langle T[\,]\rangle \\
read &: [0, num_elements\langle T\rangle - 1] \rightarrow access_helper\langle T[\,]\rangle \\
read &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow \\
& \quad type_of\langle public_member\langle T\rangle(m)\rangle \\
read &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow \\
& \quad access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
write &: T \rightarrow void \\
write &: public_member\langle T\rangle(m) \times type_of\langle public_member\langle T\rangle(m)\rangle \rightarrow void \\
write &: public_member\langle T\rangle(m) \rightarrow access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle \\
write &: [0, num_elements\langle T\rangle - 1] \times T[\,] \rightarrow void \\
write &: access_helper\langle T\rangle \times [0, num_elements\langle T\rangle - 1] \times T[\,] \rightarrow void \\
write &: access_helper\langle T\rangle \times [0, num_elements\langle T\rangle - 1] \rightarrow access_helper\langle T[\,]\rangle \\
write &: [0, num_elements\langle T\rangle - 1] \rightarrow access_helper\langle T[\,]\rangle \\
write &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \times \\
& \quad type_of\langle public_member\langle T\rangle(m)\rangle \rightarrow void \\
write &: access_helper\langle T\rangle \times public_member\langle T\rangle(m) \rightarrow
\end{aligned}
$$

mode/kind	*raw*
read	memory_read_raw_if<AWidth, DWidth>
write	memory_write_raw_if<AWidth, DWidth>
read/write	memory_read_write_raw_if<AWidth, DWidth>
mode/kind	*simple*
read	memory_read_simple_if<M.Type>
write	memory_write_simple_if<M.Type>
read/write	memory_read_write_simple_if<M.Type>
mode/kind	*array*
read	memory_read_array_if<M.Type>
write	memory_write_array_if<M.Type>
read/write	memory_read_write_array_if<M.Type>

Table 5.3: Overview of Memory Accessor Channel interfaces

$$access_helper\langle type_of\langle public_member\langle T\rangle(m)\rangle\rangle$$

□

Definition 5.6.2.8 (Memory Port):
A Memory Port, called $Port_{Mem} = [Required_{Mem_{IF}}, Bound_{Mem_{IF}}]$*, is a special Port (see Definition 5.4.2.1) with* $Bound_{Mem_{IF}} \in$ **Memory Accessor** *of the following kinds:*

Raw: *can be bound to all* $ma \in$ **Memory Accessor** *where* $ma.M.Size \geq DWidth \cdot 2^{AWidth}$

- *Read Raw Port:* $Port_{Mem_{read_raw}}\langle AWidth, DWidth\rangle$ *with* $Required_{Mem_{IF}} := memory_read_raw_if\langle AWidth, DWidth\rangle$
- *Write Raw Port:* $Port_{Mem_{write_raw}}\langle AWidth, DWidth\rangle$ *with* $Required_{Mem_{IF}} := memory_write_raw_if\langle AWidth, DWidth\rangle$
- *Read/Write Raw Port:* $Port_{Mem_{read_write_raw}}\langle AWidth, DWidth\rangle$ *with* $Required_{Mem_{IF}} := memory_read_write_raw_if\langle AWidth, DWidth\rangle$

Simple: *can be bound to all* $ma \in$ **Memory Accessor** *where* $ma.M.Type == T$

- *Read Simple Port:* $Port_{Mem_{read_simple}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_read_simple_if\langle T\rangle$
- *Write Simple Port:* $Port_{Mem_{write_simple}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_write_simple_if\langle T\rangle$
- *Read/Write Simple Port:* $Port_{Mem_{read_write_simple}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_read_write_simple_if\langle T\rangle$

Array: *can be bound to all* $ma \in$ **Memory Accessor** *where* $ma.M.Type == T$

- *Read Array Port:* $Port_{Mem_{read_array}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_read_array_if\langle T\rangle$
- *Write Array Port:* $Port_{Mem_{write_array}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_write_array_if\langle T\rangle$
- *Read/Write Array Port:* $Port_{Mem_{read_write_array}}\langle T\rangle$ *with* $Required_{Mem_{IF}} := memory_read_write_array_if\langle T\rangle$

□

5.6.2.6 Software Socket

Definition 5.6.2.9 (Software Socket):
The Software Socket is a tuple $Socket_{SW} = [Port_{clk}, Port_{rst}, Port_{RMI}, A]$*, where*

1. $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*

2. $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) reset port. When the reset signal bound to this port is* `true` *the Actor A inside the Software Socket is reset with the next rising edge of the signal bound to* $Port_{clk}$*. In this case the active leaf Behavior of Actor A is terminated immediately and the constructor of the root Behavior of A is called. When the reset signal changes back to* `false` *the computation restarts at the root Behavior's* `main` *routine with the next rising edge of the signal bound to* $Port_{clk}$*.*

3. $Port_{RMI}$ *is the single RMI Client Port of type* $Port_{RMI_C}$*. The RMI port is bound to an RMI Channel. All service calls on Ports of Actor A are routed through this RMI port.*

4. $A \in$ **Actor** *is the Actor mapped to this Software Socket. This Actor's Behavior can be of kind Active Sequential and Reactive Sequential. All EET annotations inside the Actor's Behaviors are mapped to the corresponding number of clock cycles of the reference frequency* $f_{clk_{ref}}$ *(used during timing estimation):*

$$\begin{aligned} M_{EET} &: \mathbb{D} \to \mathbb{N}_{\geq 0} \\ M_{EET}(x) &= |x| \cdot f_{clk_{ref}} \end{aligned}$$

with duration $\mathbb{D} = (value, unit)$ *of the EET and the scaling function* $|X|: unit \to \mathbb{R}^+$ *(see Section 5.5.4.5).*

□

5.6.2.7 Hardware Socket

A Hardware Socket to represent custom hardware elements of the target platform. In the Virtual Target Architecture Layer this can either be an *Actor Socket* or a *Shared Object Socket.*

Definition 5.6.2.10 (Actor Socket):
The Actor Socket is a tuple $Socket_{HW} = [Port_{clk}, Port_{rst}, \overline{Port_{RMI}}, \overline{Port_{Mem}}, \overline{Port_{Signal}}, A]$*, where*

1. $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*

2. $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) reset port. When the reset signal bound to this port is* `true` *the Actor A inside the Actor Socket is reset with the next rising edge of the signal bound to* $Port_{clk}$*. In this case* ***all*** *active leaf Behaviors of Actor A are terminated immediately and the constructor of the root Behavior of A is called. When the reset signal changes back to* `false` *the computation restarts at the root Behavior's* `main` *routine with the next rising edge of the signal bound to* $Port_{clk}$*.*

3. $\overline{Port_{RMI}}$ *is the RMI Client Port vector of type* $Port_{RMI_C}$*. The size* $|\overline{Port_{RMI}}|$ *of this vector depends on the type of Behavior B inside Actor A:*

$$|\overline{Port_{RMI}}| = \begin{cases} par_set(B) & \text{if } B.kind \in \{Active\ Parallel, Reactive\ Parallel\} \\ 1 & \text{if } B.kind \in \{Active\ Sequential, Reactive\ Sequential\} \end{cases}$$

Each element of the RMI port vector is bound to an RMI Channel[5]*. All service calls on Ports of Actor A are routed through this RMI port vector.*

[5] Multiple elements of the RMI port vector can be bound to the same RMI Channel.

4. $\overline{Port_{Mem}}$ *is a vector of ports* $p_i \in \mathbf{Port_{Mem}}$. *The Actor A can access these ports after its binding to the Socket.*

5. $\overline{Port_{Signal}}$ *is a vector of ports* $p_i \in \mathbf{Port_{Signal}}$. *The Actor A can access these ports after its binding to the Socket.*

6. $A \in$ **Actor** *is the Actor mapped to this Actor Socket. This Actor's Behavior can be of kind Active Sequential, Reactive Sequential, Active Parallel and Reactive Parallel. For the EET annotations inside the Actor's Behaviors one of the following rules needs to be applied:*

   - *mapping to the corresponding number of clock cycles of the reference frequency* $f_{clk_{ref}}$ *(as described for the Software Socket above) or*
   - *manual insertion of* `wait(N)` *boundaries per EET block with the following properties:*
     (a) $N \in \mathbb{N}_{>0}$ *is the number of clock cycles*
     (b) *for each EET block E the total number of annotated clock cyles* $\sum_{EET(E)} N \leq M_{EET}(E)$

□

Definition 5.6.2.11 (Shared Object Socket):
The Shared Object Socket is a tuple $Socket_{SO} = [Port_{clk}, Port_{rst}, \overline{Port_{RMI}}, \overline{Port_{Mem}}, \overline{Port_{Signal}}, SO]$, *where*

1. $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*

2. $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) reset port. When the reset signal bound to this port is* `true` *the Shared Object SO inside the Socket is reset with the next rising edge of the signal bound to* $Port_{clk}$. *In this case the RMI server process is reset and the constructor of SO is called. When the reset signal changes back to* `false` *the RMI server process is ready for a new RMI transaction with the next rising edge of the signal bound to* $Port_{clk}$.

3. $\overline{Port_{RMI}}$ *is the RMI Server Port vector of type* $Port_{RMI_S}$. *The size* $|\overline{Port_{RMI}}|$ *of this vector depends on the number of RMI Channels (see Definition 5.6.2.13) this Socket is bound to. For each RMI Channel a dedicated RMI Server Port is required. Each element of* $\overline{Port_{RMI}}$ *is required to be bound to a different channel.*

4. $\overline{Port_{Mem}}$ *is a vector of ports* $p_i \in \mathbf{Port_{Mem}}$. *The Shared Object SO can access these ports after its binding to the Socket.*

5. $\overline{Port_{Signal}}$ *is a vector of ports* $p_i \in \mathbf{Port_{Signal}}$. *The Shared Object SO can access these ports after its binding to the Socket.*

6. $SO \in$ **SO** *is the Shared Object mapped to this Socket. For the EET annotations inside the Shared Object Services one of the following rules needs to be applied:*

   - *mapping to the corresponding number of clock cycles of the reference frequency* $f_{clk_{ref}}$ *(as described for the Software Socket above) or*
   - *manual insertion of* `wait(N)` *boundaries per EET block with the following properties:*
     (a) $N \in \mathbb{N}_{>0}$ *is the number of clock cycles*
     (b) *for each EET block E the total number of annotated clock cyles* $\sum_{EET(E)} N \leq M_{EET}(E)$

□

RMI general term	VTA modeling element
Client	*Software Socket* (see Definition 5.6.2.9) or *Actor Socket* (see Definition 5.6.2.10)
Stub	*RMI Client Port* (see Definition 5.6.2.3)
Network	*RMI Channel* (see Definition 5.6.2.13)
Skeleton	*RMI Server Port* (see Definition 5.6.2.3)
Server	*Shared Object Socket* (see Definition 5.6.2.11)

Table 5.4: RMI general term to VTA modeling element mapping

5.6.2.8 RMI Channel

The *Remote Method Invocation* (RMI) performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized objects. RMI is based on a client server communication scheme. A client sends requests to a server and the server responds to these requests. Usually, the client and the server are two different computers which are connected through a communication network. Figure 5.56 gives an overview of the RMI client server architecture with stub and skeleton.

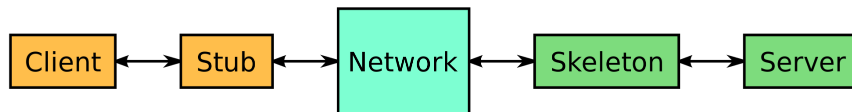


Figure 5.56: RMI client server architecture with stub and skeleton

The *stub* acts as a gateway for client side. It represents all public methods of the remote object and enables the client to call them locally on the stub. The stub transforms a client side's call into outgoing requests to the server side object. All requests and responses are routed through the network, thus enabling reliable communication between client and server.

The stub is responsible for:

- initiating the communication towards the server skeleton
- translating calls (call to object and method ID mapping)
- serialization of the parameters
- informing the skeleton about the call request
- passing arguments to the skeleton over the network
- deserialization of the response from the skeleton (i.e. return data, if any)
- informing the skeleton that the call is complete

The *skeleton* acts as gateway for server side objects. All incoming client requests are routed through it. The skeleton wraps server object functionality and exposes it to the clients.

The skeleton is responsible for:

- translating incoming data from the stub to the correct up-calls to server objects
- deserialization of the arguments from received data
- passing arguments to server objects
- serialization of the returned values from server objects
- passing values back to the client stub over the network

Table 5.4 shows the mapping of the RMI general terms to the modeling elements of the Virtual Target Architecture Layer. The RMI protocol is hidden inside the *RMI Channel*. The RMI Ports bound to the RMI Channel, which implements the RMI Client and RMI Server Interfaces inside the RMI Channel.

Definition 5.6.2.12 (RMI Interface):
The RMI Interface defines the services of the RMI Channel to call remote objects and to route and transfer the client's call requests to the Shared Object Socket (Server). The RMI Client Port can be bound to the RMI Client Interface and the RMI Server Port can be bound to the RMI Server Interface.

RMI_Client_IF *provides two services:*

1. *Remote method call with no (`void`) return value*

$$call_{procedure}\colon \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times Bitvector \rightarrow void$$
$$call_{procedure}(ClientID, ObjectID, MethodID, Parameters)$$

2. *Remote method call with return value*

$$call_{function}\colon \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times Bitvector \rightarrow Bitvector$$
$$Return_Value = call_{function}(ClientID, ObjectID, MethodID, Parameters)$$

where

- **ClientID** *is a globally unique client identifier,*
- **ObjectID** *is a globally unique object identifier (each instance of a Shared Object has this unique identifier; the Object Socket has the same identifier as its Shared Object),*
- **MethodID** *is a per object unique method identifier,*
- **Parameters** *is the serialized bitvector representation of all parameters of the called method,*
- **Return_Value** *is the serialized bitvector representation of all return parameters of the called method.*

RMI_Server_IF *provides five services:*

1. *Waiting for requests*

$$listen_for_action\colon void \rightarrow \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$$
$$[ClientID, ObjectID, MethodID] = listen_for_action()$$

The server listens for requests to serve. If a request from a client has been detected, it returns the the request's $ClientID$, $ObjectID$ and $MethodID$. IF the $ClientID$ has been registered for the targeted Shared Object Socket AND the Shared Object Socket contains a Shared Object with $ObjectID$ AND the Shared Object has a method with $MethodID$ THEN the requested method call is served.

2. *Waiting for guard evaluation and scheduling*

$$wait_for_guard\colon \mathbb{N}_{\geq 0} \times Boolean \rightarrow void$$
$$wait_for_guard(ClientID, is_busy)$$

*This service returns the status of the Shared Object's guard evaluation and scheduling. IF the method access request of ClientID has been granted THEN is_busy becomes **false**, otherwise it is **true**.*

3. *Reception of method parameters*

$$receive_in_params\colon void \rightarrow Bitvector$$
$$Parameters = receive_in_params()$$

This service receives the parameters of the remote method to be called.

4. *Waiting for method execution to be finished*

$$return_params_idle\colon \mathbb{N}_{\geq 0} \times Boolean \rightarrow void$$
$$return_params_idle(ClientID, is_busy)$$

*This service return the status of the remote method call of ClientID after the method parameters have been received. IF the execution of the method has been finished THEN is_busy becomes **false**, otherwise it is **true**.*

5. *Provide method's return parameters*

$$provide_return_params\colon \mathbb{N}_{\geq 0} \times Bitvector \rightarrow void$$
$$provide_return_params(ClientID, Return_Value)$$

This service provides the return parameters after completion of the requested method call of ClientID.

The implementation of these Interfaces and the mapping of these interface services to the chosen communication network is performed by the RMI Channel. □

Definition 5.6.2.13 (RMI Channel):
The Remote Method Invocation (RMI) Channel is a tuple

$$RMI_Channel = [Port_{clk}, Port_{rst}, \overline{Bindings}, OSSS_Channel, Start_Address]$$

where

- $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*
- $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the (active high) reset port. When the reset signal bound to this port is* `true` *the RMI Channel is reset with the next rising edge of the signal bound to* $Port_{clk}$*. When the reset signal changes back to* `false` *the RMI Channel is ready for new RMI calls with the next rising edge of the signal bound to* $Port_{clk}$*.*
- $\overline{Bindings}$ *is a list of tuples* $(Client, Server)$ *with* $Client \in \mathbf{Port_{RMI_C}}$ *and* $Server \in \mathbf{Port_{RMI_S}}$*. This list represents the Application Layer Communication Links between Actor Ports and Shared Objects. The length of the list, written as* $l_{Bindings_{RMI}} = |\overline{Bindings}|$ *represents the number of maximal parallel method requests on the RMI Channel. Based on the connection topology we define the following constraints:*
  - *bus topology:* $l_{Bindings_{RMI}} \geq 1$
  - *point-to-point topology:* $l_{Bindings_{RMI}} = 1$
- $OSSS_Channel \in$ **OSSSChannel** *is the transport channel inside the RMI channel. For our RMI Channels this need to be an OSSS Channel (see Definition 5.6.2.14) implementing a shared bus or point-to-point topology.*
- $Start_Address \in \mathbb{N}_{\geq 0}$ *is the absolute start address of all RMI servers's (Shared Object Socket) Method ID, status, message, argument and return registers which are connected to this RMI channel (see Figure 5.57 for the example of a memory layout for an Object Socket with N-1 clients and 32 bit bus data width). For P2P Channels this value has no effect.*

An RMI Channel provides the following helper functions to map unique client and object IDs to the RMI Channel's address space and vice versa:

Object ID to Address:

$$\begin{aligned} map_{OID_to_ADDR} &: \mathbb{N}_{\geq 0} \rightarrow \mathbb{Z} \\ Base_Addr &= map_{OID_to_ADDR}(Object_ID) \end{aligned}$$

Returns the base address (≥ Start_Address) of the address space for an object ID. Returns -1 if the object ID is invalid.

Client ID to Address:

$$\begin{aligned} map_{CID_to_ADDR} &: \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \to \mathbb{Z} \\ Client_Register_Base_Addr &= map_{CID_to_ADDR}(Object_ID,\ Client_ID) \end{aligned}$$

Returns the base address of the client register for a given object and client ID. Returns -1 if either the object or client ID is invalid.

Object ID to Arguments:

$$\begin{aligned} map_{OID_to_ARG} &: \mathbb{N}_{\geq 0} \to \mathbb{Z} \\ Client_Register_Base_Addr &= map_{OID_to_ARG}(Object_ID) \end{aligned}$$

Returns the base address of the arguments register for a given object ID. Returns -1 if the provided object ID is invalid.

Object ID to Return Value:

$$\begin{aligned} map_{OID_to_RET} &: \mathbb{N}_{\geq 0} \to \mathbb{Z} \\ Client_Register_Base_Addr &= map_{OID_to_RET}(Object_ID) \end{aligned}$$

Returns the base address of the return value register for a given object ID. Returns -1 if the provided object ID is invalid.

Address to Object ID:

$$\begin{aligned} map_{ADDR_to_OID} &: \mathbb{N}_{\geq 0} \to \mathbb{Z} \\ Object_ID &= map_{ADDR_to_OID}(Addr) \end{aligned}$$

Returns the object ID that belongs to the given address. If no registers of the provided address belong to any object ID the value -1 is returned.

Address to Client ID:

$$\begin{aligned} map_{ADDR_to_CID} &: \mathbb{N}_{\geq 0} \to \mathbb{Z} \times \mathbb{Z} \\ \langle Object_ID,\ Client_ID \rangle &= map_{ADDR_to_CID}(Addr) \end{aligned}$$

Returns the object and client IDs that belongs to the given address. If no registers of the provided address belong to any object ID the value $\langle -1, -1 \rangle$ is returned. If the address belongs to the registers of a valid object, but does not point within the status or message register of a client, the value $\langle OID, -1 \rangle$ is returned.

The size of the Argument and Return Value registers have the maximums sizes to hold the largest parameter list and return value of the corresponding object.

$$\begin{aligned} size_{Arguments} &: \mathbb{N}_{\geq 0} \to \mathbb{N}_{\geq 0} \\ size_{Arguments}(OID) &= \left\lceil \frac{\max_{mid=0}^{num_mid(OID)-1} \{ sum_{j=0}^{num_param(mid)-1} size(param_j) \})}{bus_addr_width} \right\rceil \end{aligned}$$

Method IDs are numbered $0, 1, 2, \ldots$ for each object. $num_mid \colon \mathbb{N}_{\geq 0} \to \mathbb{N}_{>0}$ provides the total number of methods for each object ID. Parameters are numbered $0, 1, 2, \ldots$ for each method. $num_param \colon \mathbb{N}_{\geq 0} \to \mathbb{N}_{\geq 0}$ provides the total number of parameters for each method ID. $size \colon mathbbN_{\geq 0} \to \mathbb{N}_{\geq 0}$ provides the bit-width of a parameter. bus_addr_width is the bit-with of the shared bus inside the RMI channel, usually in multiples of 8 bit, e.g. $8, 16, 32, 64, \ldots$ bit.

$$\begin{aligned} size_{Return} &: \mathbb{N}_{\geq 0} \to \mathbb{N}_{\geq 0} \\ size_{Return}(OID) &= \left\lceil \frac{\max_{mid=0}^{num_mid(OID)-1} \{ size(return_val_{mid}) \}}{bus_addr_width} \right\rceil \end{aligned}$$

Same as $size_{Arguments}$ but returns the total bit-with of the Return Value register. □

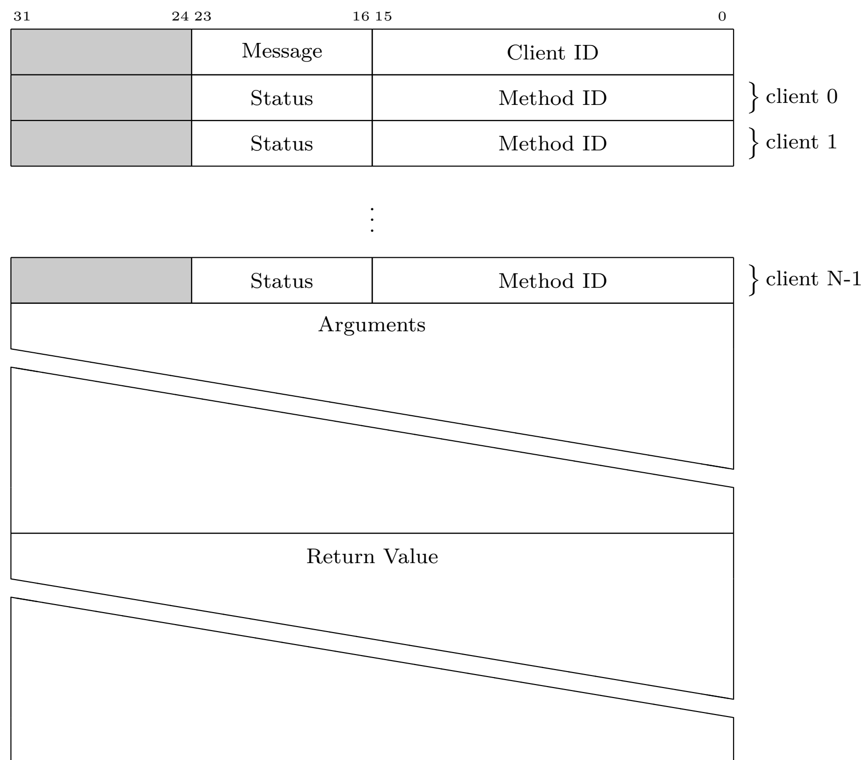


Figure 5.57: Memory layout (big-endian) for an Object Socket with N-1 clients and 32 bit bus data width

Bit field	ID	Name	Description
00000000	0	idle	either no call requested or last call successfully completed (i.e. all return values have been read)
00000001	1	requested	method call requested (see "*Method ID*" field)
00000010	2	arg_ready	all method arguments have been written into "*Arguments*" registers
00000011	3	return_ready	method has been executed, return value is available in "*Return Value*" registers

Table 5.5: RMI client status register organization

Bit field	ID	Name	Description
00000000	0	no message	-
00000001	1	call granted	requested call has been granted (see "*Method ID*" field for client whose request has been granted)
00000010	2	return value ready	requested call has been completely executed and the return value is available (see "*Method ID*" field for client whose call has been completed)

Table 5.6: RMI Object Socket message register organization

Figure 5.57 shows an example of the memory layout for an Object Socket with N-1 clients and 32 bit bus data width. Figure 5.58 gives an overview of the phases of RMI client and server using the described memory layout to communicate and synchronize. Figure 5.59 provides a more detailed view on the client and server RMI state machines and their interaction.

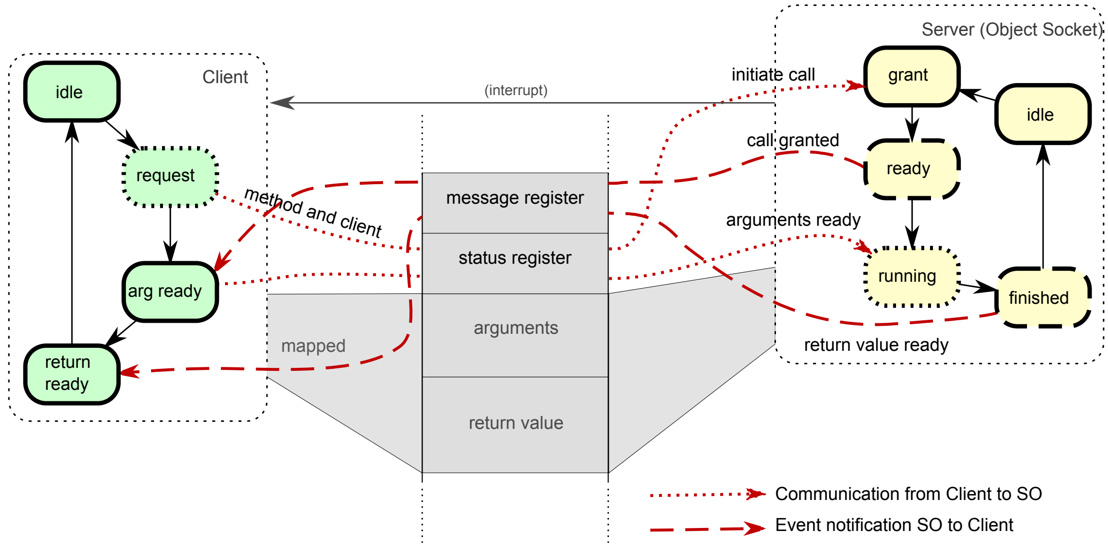


Figure 5.58: Phases of client and server mapped to RMI Channel address layout (based on [17])

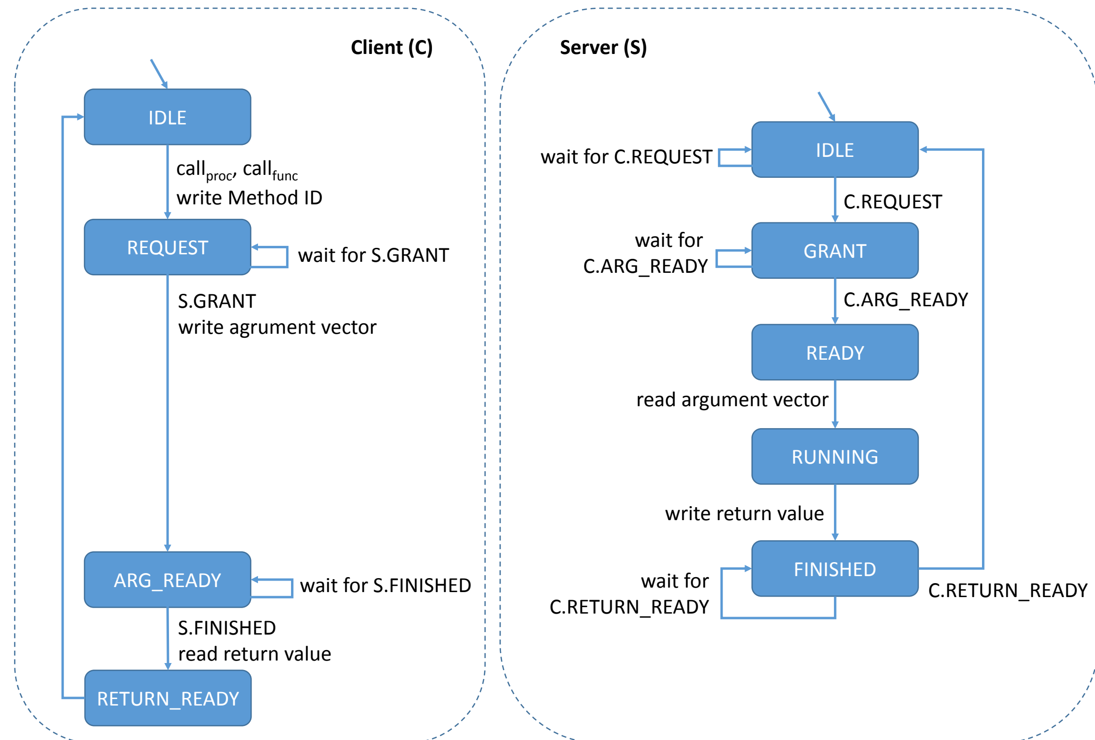


Figure 5.59: Abstract interaction of RMI Client and Server State Machines

Further details on the operational semantics can be found in Section 5.6.4. For implementation details of the RMI protocol see Section 6.5.2.

5.6.2.9 OSSS Channel

Definition 5.6.2.14 (OSSS Channel):
The OSSS Channel is a tuple

$$\begin{aligned} OSSS_Channel = [&Port_{clk}, Port_{rst} \\ &master, slave, \\ &\overline{data_width_{master}}, \overline{data_width_{slave}}, \\ &addr_width, addr_decoder, \\ &arbiter] \end{aligned}$$

where

1. $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in \textbf{Signal}$ *is the clock port. When using an OSSS Channel inside an RMI Channel, the clock signal of the OSSS Channel is derived form the clock signal of the RMI Channel.*

2. $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in \textbf{Signal}$ *is the (active high) reset port. When the reset signal bound to this port is* `true` *the OSSS Channel is reset with the next rising edge of the signal bound to* $Port_{clk}$*. When the reset signal changes back to* `false` *the OSSS Channel is ready for new data transfers with the next rising edge of the signal bound to* $Port_{clk}$*. When using an OSSS Channel inside an RMI Channel, the reset signal of the OSSS Channel is derived form the reset signal of the RMI Channel.*

3. $master \in \mathbb{N}_{>0}$ *is the number of channel masters (also called "initiators") that initiate any data transfers on the channel. Each master has a unique identifier, for* $master = M$ *this would be* $master_0 = 1, master_1 = 2, \ldots, master_{M-1} = M$.

4. $slave \in \mathbb{N}_{>0}$ *is the number of channel slaves (also called "targets") that react onto requests from masters. Each slave has a unique identifier, for* $slave = S$ *this would be* $slave_0 = 1, slave_1 = 2, \ldots, slave_{S-1} = S$.

5. $\overline{data_width_{master}} \in \mathbb{N}_{>0}^{master}$ *is a vector of size master with elements of type* $\mathbb{N}_{>0}$*. It defines the data width (in bit) of each master in the channel.*

6. $\overline{data_width_{slave}} \in \mathbb{N}_{>0}^{slave}$ *is a vector of size slave with elements of type* $\mathbb{N}_{>0}$*. It defines the data width (in bit) of each slave in the channel.*

7. $addr_width \in \mathbb{N}_{>0} \cup \bot$ *is the channel's address width in bits. The symbol* $\bot$ *is used when the channel has no address lines.*

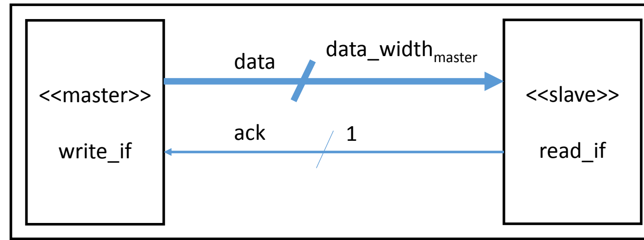
8. $addr_decoder$ *is a function* $\mathbb{N}_{\geq 0} \rightarrow Boolean^{slave}$ *that maps an address from the global address space into the slave's address space, represented as tuple* $[base_addr, high_addr]$ *with* $base_addr, high_addr \in \mathbb{N}_{\geq 0} \wedge base_addr < high_addr$*. Address spaces of different slaves cannot overlap. When the address argument of the mapping function* $addr_decoder$ *is within the address space of one of the channel's slaves, the Boolean output vector gets "true" on the index of the corresponding slave. Otherwise the Boolean output vector is "false" for all its elements.*

9. $arbiter$ *is a function* $Boolean^{master} \rightarrow \mathbb{N}_{>0}$ *that grants, in the case of multiple masters, channel access for a single master. The arbiter function selects, based on the provided scheduling algorithm, one of the masters from the request vector. The OSSS Channel's arbiter function is equivalent to the Shared Object's scheduler function (see Section 5.5.2.3). For OSSS Channels the same pre-defined scheduling algorithms can be used (see Section 5.5.3).*

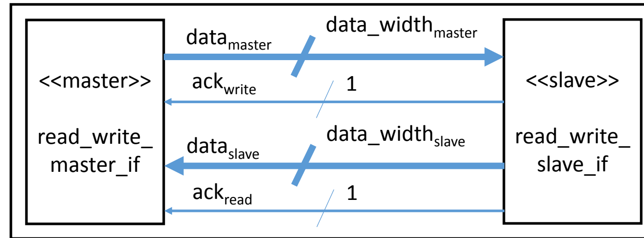
The OSSS Channel supports two different topologies: Point-to-Point and Shared Bus connection, with the following properties:

Point-to-Point (P2P) *A Point-to-Point channel is defined as* $OSSS_Channel_{P2P} = [clk, rst, data_width_{master}, data_width_{slave}]$. *It is a specialized OSSS Channel with the following constraints:*

- $OSSS_Channel.Port_{clk} := clk$
- $OSSS_Channel.Port_{rst} := rst$
- $OSSS_Channel.master := 1$
- $OSSS_Channel.slave := 1$
- $OSSS_Channel.\overline{data_width_{master}} := data_width_{master}$
- $OSSS_Channel.\overline{data_width_{slave}} := \begin{cases} 0 & \text{if unidirectional} \\ data_width_{slave} & \text{if bidirectional} \end{cases}$
- $OSSS_Channel.addr_width := \perp$
- $OSSS_Channel.addr_decoder := \emptyset$
- $OSSS_Channel.arbiter := \emptyset$



(a) unidirectional P2P channel



(b) bidirectional P2P channel

Figure 5.60: Usage of point-to-point channel interfaces for uni- and bidirectional connections

The $OSSS_Channel_{P2P}$ *implements a read and a write interface. The master side can be used through the write_if and the slave side can be used through the read_if:*

write_if *is the master side interface used to implement an unidirectional point-to-point channel. In this case* $data_width_{slave} := 0$ *It consists of the following services:*

1. *Sending plain data from the master to the salve:*

$$\begin{aligned} write_blocking &: Bitvector_{data_width_{master}} \rightarrow Boolean \\ success &= write_blocking(data_chunk) \end{aligned}$$

   Sends a Bitvector of size $data_width_{master}$ *to the slave. The function returns "true" if the transfer has been successfully completed*

2. *Sending a Serializable Object from the master to the slave:*

$$write_blocking : Serializable_Object \rightarrow Boolean$$

$$success = write_blocking(ser_obj)$$

Sends a Serializable Object (which can be of different sizes) to the slave. The function returns "true" if the transfer has been successfully completed.

3. *Sending a set of Serializable Objects (called Serializable Archive) from the master to the slave:*

$$\begin{aligned} write_blocking &: Serializable_Archive \to Boolean \\ success &= write_blocking(ser_arch) \end{aligned}$$

Sends a Serializable Archive (which consists of multiple Serialized Objects) to the slave. The function returns "true" if the transfer has been successfully completed.

read_if *is the slave side interface used to implement an unidirectional point-to-point channel. In this case* $data_width_{slave} := 0$ *It consists of the following services:*

1. *Slave receives plain data from the master:*

$$\begin{aligned} read_blocking &: void \to Bitvector_{data_width_{master}} \\ data_chunk &= read_blocking() \end{aligned}$$

2. *Slave receives a Serializable Object from the master:*

$$\begin{aligned} read_blocking &: void \to Serializable_Object \\ ser_obj &= read_blocking() \end{aligned}$$

3. *Slave receives a set of Serializable Objects (called Serializable Archive) from the master:*

$$\begin{aligned} read_blocking &: void \to Serializable_Archive \\ ser_arch &= read_blocking() \end{aligned}$$

read_write_master_if *is the master side interface used to implement a bidirectional point-to-point channel. It consists of the following services:*

$$\begin{aligned} write_blocking &: Bitvector_{data_width_{master}} \to Boolean \\ write_blocking &: Serializable_Object \to Boolean \\ write_blocking &: Serializable_Archive \to Boolean \\ read_blocking &: void \to Bitvector_{data_width_{slave}} \\ read_blocking &: void \to Serializable_Object \\ read_blocking &: void \to Serializable_Archive \end{aligned}$$

read_write_slave_if *is the slave side interface used to implement a bidirectional point-to-point channel. It consists of the following services:*

$$\begin{aligned} write_blocking &: Bitvector_{data_width_{slave}} \to Boolean \\ write_blocking &: Serializable_Object \to Boolean \\ write_blocking &: Serializable_Archive \to Boolean \\ read_blocking &: void \to Bitvector_{data_width_{master}} \\ read_blocking &: void \to Serializable_Object \\ read_blocking &: void \to Serializable_Archive \end{aligned}$$

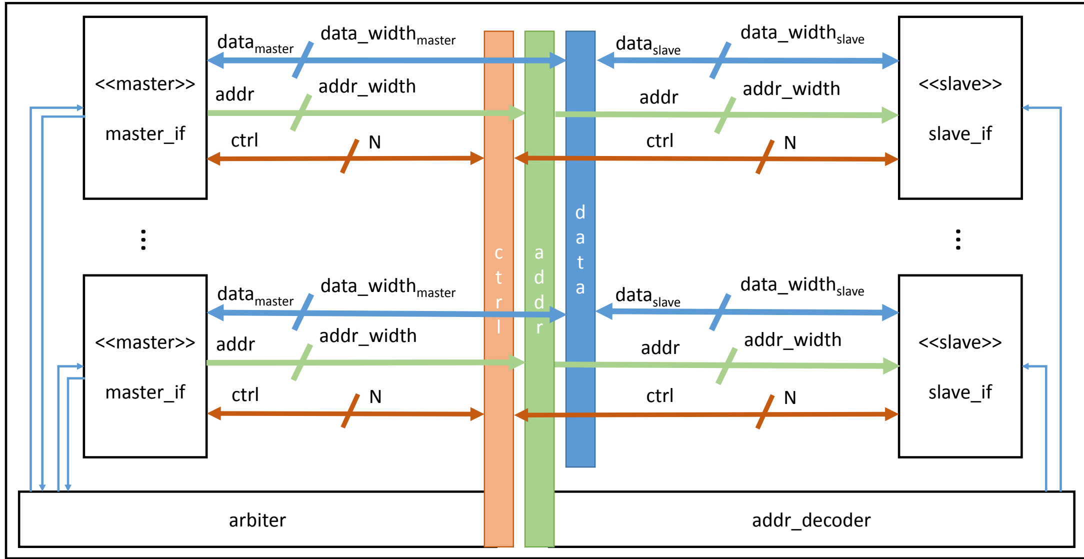


Figure 5.61: Usage of master, slave, arbiter and address decoder for modeling shared buses

Figure 5.60 gives an overview about a possible usage of point-to-point channel interfaces for uni- and bidirectional connections.

Shared Bus *A Shared Bus is defined as*

$$\begin{aligned} OSSS_Channel_{Bus} = [&clk, rst \\ &master, slave, \\ &\overline{data_width_{master}}, \overline{data_width_{slave}}, \\ &addr_width, addr_decoder, \\ &arbiter] \end{aligned}$$

It is a specialized OSSS Channel with the following constraints:

- $OSSS_Channel.Port_{clk} := clk$
- $OSSS_Channel.Port_{rst} := rst$
- $OSSS_Channel.master := master$
- $OSSS_Channel.slave := slave$
- $OSSS_Channel.\overline{data_width_{master}} := \overline{data_width_{master}}$ *with* $\forall i \in \{0, \ldots, master - 1\}\colon data_width_{master}[i] \mod 8 = 0$
- $OSSS_Channel.\overline{data_width_{slave}} := \overline{data_width_{slave}}$ *with* $\forall i \in \{0, \ldots, slave - 1\}\colon data_width_{slave}[i] \mod 8 = 0$
- $data_width_{master}[i] \geq data_width_{slave}[j]$ $\forall i, j\ i \in \{0, \ldots, master - 1\}, j \in \{0, \ldots, slave - 1\}$
- $OSSS_Channel.addr_width := addr_width$ *with* $addr_width \mod 8 = 0$
- $OSSS_Channel.addr_decoder := addr_decoder$
- $OSSS_Channel.arbiter := \begin{cases} \emptyset & \text{if } master == 1 \\ arbiter & \text{if } master > 1 \end{cases}$

The $OSSS_Channel_{Bus}$ *implements a master and a slave interface. The master side can be used through the master_if and the slave side can be used through the slave_if:*

master_if *implements the interface of a bus master and consists of the following services:*

1. *writing plain data, a Serializable Object or a Serializable Archive from a master to a slave:*

$$\begin{aligned} write_blocking &: \mathbb{N}_{\geq 0} \times Bitvector_{data_width_{master}} \times Boolean \rightarrow Boolean \\ success &= write_blocking(slave_addr, data_chunk, burst) \end{aligned}$$

$$\begin{aligned} write_blocking &: \mathbb{N}_{\geq 0} \times Serializable_Object \times Boolean \rightarrow Boolean \\ success &= write_blocking(slave_addr, ser_obj, burst) \end{aligned}$$

$$\begin{aligned} write_blocking &: \mathbb{N}_{\geq 0} \times Serializable_Archive \times Boolean \rightarrow Boolean \\ success &= write_blocking(slave_addr, ser_arch, burst) \end{aligned}$$

2. *reading plain data, a Serializable Object or a Serializable Archive from a slave:*

$$\begin{aligned} read_blocking &: \mathbb{N}_{\geq 0} \times Boolean \rightarrow Boolean \times Bitvector_{data_width_{master}} \\ \langle success, data_chunk \rangle &= read_blocking(slave_addr, burst) \end{aligned}$$

$$\begin{aligned} read_blocking &: \mathbb{N}_{\geq 0} \times Boolean \rightarrow Boolean \times Serializable_Object \\ \langle success, ser_obj \rangle &= read_blocking(slave_addr, burst) \end{aligned}$$

$$\begin{aligned} read_blocking &: \mathbb{N}_{\geq 0} \times Boolean \rightarrow Boolean \times Serializable_Archive \\ \langle success, ser_arch \rangle &= read_blocking(slave_addr, burst) \end{aligned}$$

slave_if *implements the interface of a bus slave and consists of the following services:*

1. *waits for requests from the master and return the requested service (read or write) and address:*

$$\begin{aligned} wait_for_action &: void \rightarrow \mathbb{N}_{\geq 0} \times \{read, write\} \\ \langle address, action_type \rangle &= wait_for_action() \end{aligned}$$

2. *writes plain data, a Serializable Object or a Serializable Archive to the master:*

$$\begin{aligned} write_blocking &: Bitvector_{data_width_{slave}} \rightarrow void \\ write_blocking(data_chunk) & \end{aligned}$$

$$\begin{aligned} write_blocking &: Serializable_Object \rightarrow void \\ write_blocking(ser_obj) & \end{aligned}$$

$$\begin{aligned} write_blocking &: Serializable_Archive \rightarrow void \\ write_blocking(ser_arch) & \end{aligned}$$

3. *reads plain data, a Serializable Object or a Serializable Archive from the master:*

$$\begin{aligned} read_blocking &: void \rightarrow Bitvector_{data_width_{slave}} \\ data_chunk &= write_blocking() \end{aligned}$$

$$\begin{aligned} read_blocking &: void \rightarrow Serializable_Object \\ ser_obj &= write_blocking() \end{aligned}$$

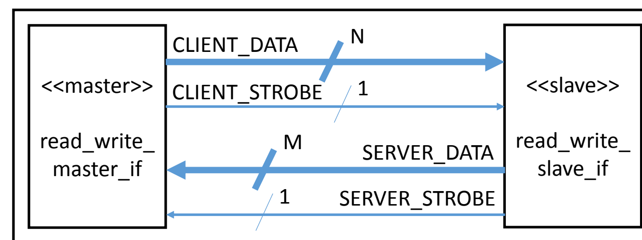
$$\begin{aligned} read_blocking &: void \rightarrow Serializable_Archive \\ ser_arch &= write_blocking() \end{aligned}$$

Figure 5.61 gives an overview about possible usage of master, slave, arbiter and address decoder interfaces for modeling shared buses.

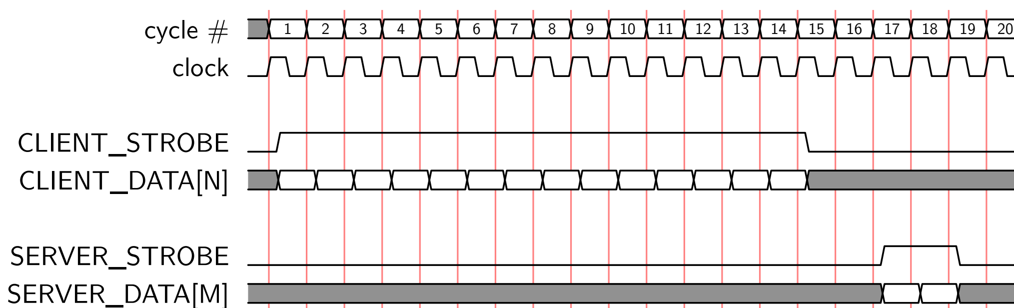
□

Definition 5.6.2.15 (Simple P2P Channel):
The Simple Point-to-Point channel $OSSS_Channel_{SimpleP2P} = [clk, rst, N, M]$ *is a pre-defined OSSS Channel optimized for the RMI protocol. Essentially it is an OSSS Channel with the following constraints:*

- $OSSS_Channel.Port_{clk} := clk$
- $OSSS_Channel.Port_{rst} := rst$
- $OSSS_Channel.master := 1$
- $OSSS_Channel.slave := 1$
- $OSSS_Channel.\overline{data_width_{master}} := N$
- $OSSS_Channel.\overline{data_width_{slave}} := M$
- $OSSS_Channel.addr_width := \perp$
- $OSSS_Channel.addr_decoder := \emptyset$
- $OSSS_Channel.arbiter := \emptyset$



(a) internal structure



(b) timing diagram

Figure 5.62: Simple point-to-point channel

Figure 5.62 gives an overview of the internal structure and the timing of the simple P2P channel. The channel is bidirectional. The master interface at the client side is capable of streaming data to the slave/server and receive data from the slave/server. The `STROBE` *signals are used to indicate the availability of data, as well as the end of data stream. Client and server data transfer can never overlap in time. The bit width N of* `CLIENT_DATA` *and M of* `SERVER_DATA` *can be assigned independently from each other.* □

Definition 5.6.2.16 (Simple Bus):
The Simple Bus

$$\begin{aligned} OSSS_Channel_{SimpleBus} = [&clk, rst,\\ &master, slave,\\ &data_width,\\ &addr_width, addr_decoder,\\ &arbiter] \end{aligned}$$

supports a limited subset of the features of the IBM OPB bus [151]. It is an OSSS Channel with the following constraints:

- $OSSS_Channel.Port_{clk} := clk$
- $OSSS_Channel.Port_{rst} := rst$
- $OSSS_Channel.master := master$
- $OSSS_Channel.slave := slave$
- $OSSS_Channel.data_width_{master}[i] := data_width \, \forall i \in \{0, \ldots, master - 1\}$
- $OSSS_Channel.data_width_{slave}[j] := data_width \, \forall j \in \{0, \ldots, slave - 1\}$
- $data_width \mod 8 = 0$
- $OSSS_Channel.addr_width := addr_width$ *with* $addr_width \mod 8 = 0$
- $OSSS_Channel.addr_decoder := addr_decoder$
- $OSSS_Channel.arbiter := \begin{cases} \emptyset & \textit{if } master == 1 \\ arbiter & \textit{if } master > 1 \end{cases}$

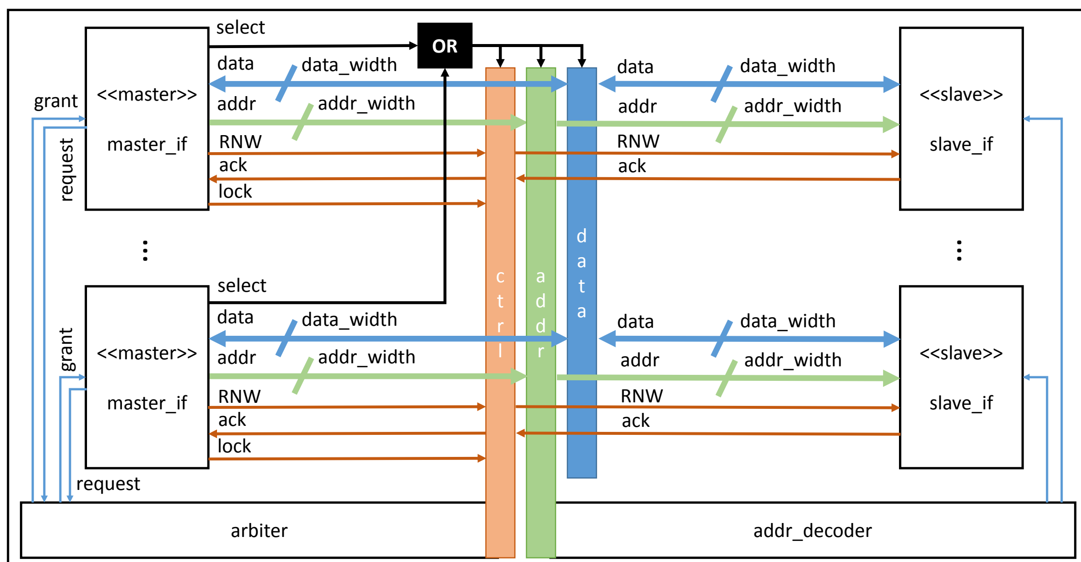


Figure 5.63: Internal structure of the simple bus

Figure 5.63 provides an overview of the internal structure of the simple bus. All masters are directly connected to the arbiter with a `request` *and grant signal. Upon request, the arbiter decides which master is allowed to use the bus. When bus access is granted, the master can use its* `select` *signal to put data, address and control information on the bus matrix. The address decoder is connected with the address bus and enables the corresponding slave, mapped into the specific address space.*

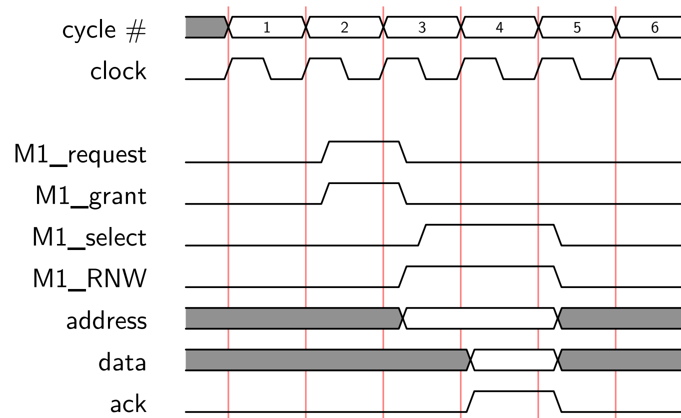


Figure 5.64: Simple bus: basic data transfer

Basic arbitration *The bus arbitration process proceed by the following protocol:*

1. *A master asserts its* `request` *signal.*
2. *The arbiter receives the request and outputs an individual* `grant` *signal to each master according to its scheduling algorithm.*
3. *The bus master samples its* `grant` *signal at the rising edge of* `clock`*. Upon grant the master may then initiate a data transfer with a slave by asserting its* `select` *signal.*

   The bus `grant` *signal is only issued by the arbiter during a valid* bus arbitration cycle*, defined as either:*

   - Idle*, which means that* `select` *and* `bus_lock` *are deasserted, indicating no data transfer is in progress, or*
   - Overlapped arbitration cycle*, which means that* `ack` *is asserted, indicating the final cycle in a data transfer, and* `bus_lock` *is not asserted. Arbitration in this cycle allows another master to begin a transfer in the following cycle, avoiding the need for a dead cycle on the bus.*

Basic data transfer *Figure 5.64 shows a typical simple bus data transfer cycle. In this example master 1 reads data from a slave. The slave has a two-cycle latency. The master asserts its* `M1_request` *signal to acquire bus access. The arbiter assert the master's grant signal according to the bus arbitration protocol, and during a valid bus arbitration cycle. After sampling the* `M1_grant` *signal at the rising edge of the* `clock` *the master asserts its* `M1_select` *signal to indicate bus ownership. The slave completes the transfer by asserting* `ack`*, which causes the causes the master to latch data from the data bus on read transfer and deassert* `M1_select`*.*

Burst data transfer *If a master asserts the* `bus_lock` *signal upon assuming control of the bus, the arbiter will continue to grant the bus to the master which locked it. Grant signals will be generated if the master asserts its* `request` *signal, during a valid arbitration cycle. Bus requests and grant signals have no effect on bus arbitration, and the master which asserted the* `bus_lock` *signal will retain control of the bus until* `bus_lock` *is deasserted for at least one complete cycle.*

Figure 5.65 shows an example where master 1 requires three non-interruptable cycles of data transfer. The slave in this example has one cycle data transfer latency. The `M1_bus_lock` *is asserted together with* `M1_select` *signal. The master may proceed with data transfer cycles while asserting* `M1_bus_lock` *without engaging in bus arbitration. The arbiter detects the* `bus_lock` *signal and continues to grant the bus to the current master, regardless of othzer (high priority) requests. This continuous data transfer is called* burst data transfer*.*

Multiple arbitration requests *Figure 5.66 shows multiple bus requests. Both masters 1 and 2 simultaneously request the bus. Master 1 has a higher priority and receives the first grant.*

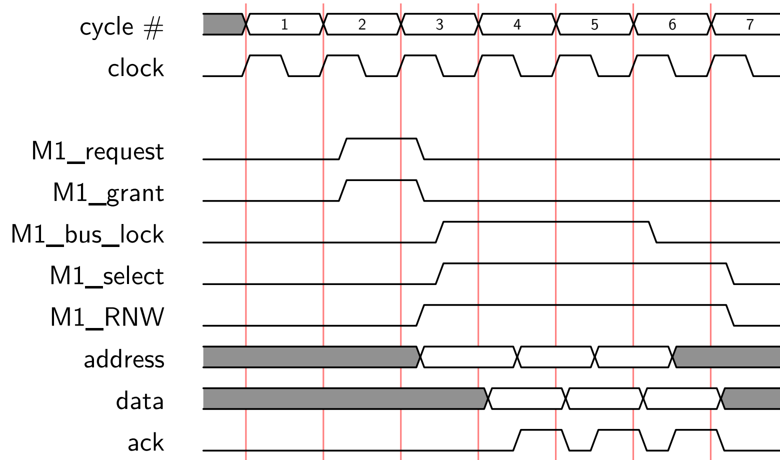


Figure 5.65: Simple bus: burst data transfer

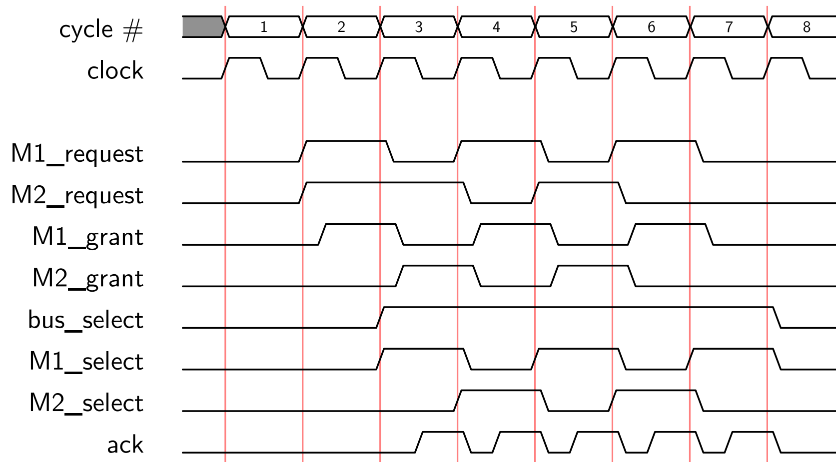


Figure 5.66: Simple bus: multiple arbitration requests

During cycle 3 master 1 completes its first transaction and master 2 received the grant for cycle 4. Thus during cycle 3 the arbitration for the bus is overlapped with a data. This improves the bandwidth of the bus.

□

5.6.2.10 IP Component

Definition 5.6.2.17 (IP Component):
An IP Component is a tuple $IPComponent = [Port_{clk}, Port_{rst}, \overline{Port}]$, *where*

- $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port.*
- $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is a synchronous, active high reset port.*
- $\overline{Port}$ *is a vector of signal ports* $Port_{Signal}$.

□

5.6.2.11 Virtual System on Chip

Definition 5.6.2.18 (Virtual Target Architecture (VTA)):
A Virtual Target Architecture (VTA) is defined as

$$\begin{aligned} VTA = [&Port_{clk}, Port_{rst}, \overline{Port_{Signal}}, \\ &\overline{Signal}, \\ &\overline{SoftwareSocket}, \\ &\overline{HardwareSocket}, \\ &\overline{Memory}, \\ &\overline{RMIChannel}, \\ &\overline{IPComponent}, \\ &\overline{Binding}] \end{aligned}$$

, where

- $Port_{clk}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is the clock port of the VTA.*
- $Port_{rst}$ *with* $Required_{IF} := signal_in_if < bool >$ *and* $Bound_{IF} \in$ **Signal** *is a synchronous, active high reset port of the VTA.*
- $\overline{Port_{Signal}}$ *is a vector of Signal Ports* $p_i \in \mathbf{Port_{Signal}}$*. These signals represent the pins of the Virtual System on Chip.*
- $\overline{Signal}$ *is a vector of Signals* $s_i \in$ **Signal***. These Signals are used to connect Signal Ports of Hardware Sockets and IP Components.*
- $\overline{SoftwareSocket}$ *is a vector of Software Socket instances* $sws_i \in \mathbf{Socket_{SW}}$*.*
- $\overline{HardwareSocket}$ *is a vector of Hardware Socket instances* $hws_i \in \mathbf{Socket_{HW}} \cup \mathbf{Socket_{SO}}$*. A Hardware Socket can either be a Shared Object Socket or an Actor Socket.*
- $\overline{Memory}$ *is a vector of Memory instances* $mem_i \in$ **Memory***.*
- $\overline{RMIChannel}$ *is a vector of RMI Channel instances* $rmi_i \in$ **RMIChannel***.*
- $\overline{IPComponent}$ *is a vector of IP Component instances* $ip_i \in$ **IPComponent***.*
- $\overline{Binding}$ *is a binding vector consisting of tuples/triples*

$$\begin{aligned} b_i \in &\{(cp, vtap) \mid cp \in C.Port_{clk} \cup C.Port_{rst}, vtap \in Port_{clk} \cup Port_{rst}\} \times \\ &\{(hwsp, vtap) \mid cp \in HWS.\overline{Port_{Signal}}, vtap \in \overline{Port_{Signal}}\} \times \\ &\{(hwsp, vtas) \mid hwsp \in HWS.\overline{Port_{Signal}}, vtas \in \overline{Signal}\} \times \\ &\{(rmip, rmich) \mid rmip \in RMIC.\overline{Port_{RMI}}, rmich \in \overline{RMIChannel}\} \times \\ &\{(hwmp, ma, mem) \mid hwmp \in HWS.\overline{Port_{Mem}}, \\ &\qquad ma \in \mathbf{MemoryAccessor}, \\ &\qquad mem \in \overline{Memory}\} \end{aligned}$$

with

$$\begin{aligned} C &\in \overline{SoftwareSocket} \cup \overline{HardwareSocket} \cup \\ &\quad \overline{Memory} \cup \overline{RMIChannel} \cup \overline{IPComponent} \\ HWS &\in \overline{HardwareSocket} \\ RMIC &\in \overline{SoftwareSocket} \cup \overline{HardwareSocket} \end{aligned}$$

and represents the following bindings:

Clock and Reset Ports *of all Architecture Objects (Software Sockets, Hardware Sockets, Memories, RMI Channels and IP Components) are bound to* $Port_{clk}$ *and* $Port_{rst}$ *of the VTA using the* $(cp, vtap)$ *tuple.*

Signal Ports *of Hardware Sockets (Shared Object Sockets and Actor Sockets) and IP Components are bound to*

1. *Signal Ports (Pins)* $\overline{Port_{Signal}}$ *of the VTA using the* $(hwsp, vtap)$ *tuple, thus establishing a connection to the environment.*
2. *Signals* $\overline{Signal}$ *of the VTA using the* $(hwsp, vtas)$ *tuple, thus establishing an internal wiring of Signal Ports of Shared Object Sockets or Actor Sockets with Signal Ports of IP Components.*

RMI Ports *of Software Sockets and Hardware Sockets (Shared Object Sockets and Actor Sockets) are bound to RMI Channels using the* $(rmip, rmich)$ *tuple.*

Memory Ports *of Hardware Sockets (Shared Object Sockets and Actor Sockets) are bound to Memories. Since a Memory Port cannot be bound directly to a Memory Component, a Memory Accessor Channel is required as proxy using the* $(hwmp, ma, mem)$ *triple.*

□

Definition 5.6.2.19 (Virtual System on Chip):
A Virtual System on Chip is defined as $VSoC = [ALS, VTA, \overline{Mapping}]$ *and consists of*

1. ALS *is an Application Layer System, as defined in Definition 5.5.2.2.*
2. VTA *is a Virtual Target Architecture, as defined in Definition 5.6.2.18.*
3. $\overline{Mapping}$ *is a mapping vector consisting of tuples*

$$\begin{aligned} m_i \in &\{(a, ss) \mid a \in ALS.\overline{Actor}, ss \in VTA.\overline{SoftwareSocket}\} \times \\ &\{(a, as) \mid a \in ALS.\overline{Actor}, as \in VTA.\overline{HardwareSocket}|_{Socket_{HW}}\} \times \\ &\{(so, sos) \mid so \in ALS.\overline{Channel}, sos \in VTA.\overline{HardwareSocket}|_{Socket_{SO}}\} \end{aligned}$$

An Actor in $ALS.\overline{Actor}$ *can either be mapped*

- *on a Software Socket in* $VTA.\overline{SoftwareSocket}$ *or*
- *on a Actor Socket in* $VTA.\overline{HardwareSocket}|_{Socket_{HW}}$.

A Shared Object in $ALS.\overline{Channel}$ *can only be mapped on a Shared Object Socket in* $VTA.\overline{HardwareSocket}|_{Socket_{SO}}$. *All Actors and Shared Object can only be mapped once, i.e. the mapping relation is bijective (one-to-one correspondence).*

□

5.6.3 Mapping rules

For applying the mapping of Actors and Shared Objects as described in Definition 5.6.2.19 the following preparations and refinements need to be done.

5.6.3.1 Add support for object serialization

To enable the transfer of user defined objects via RMI or to access user defined objects in memory components the following user-defined classes need to become serializable:

Parameters and return values of Guarded Methods $\forall SO_i \in VSoC.ALS.\overline{Channel}$ the following parameter types $T_0, \ldots T_n$ and return types T of Guarded Methods $SO_i.Method|_{GM}$ need to become serializable

$$\textbf{guarded}\ name \quad : \quad \langle guard\colon SO_i.State \rightarrow \textbf{Boolean}, \textbf{void} \rightarrow T\rangle$$

$$
\begin{array}{lcl}
\textbf{guarded } \mathit{name} & : & \langle guard\colon \textbf{true}, \textbf{void} \rightarrow T\rangle \\
\textbf{guarded } \mathit{name} & : & \langle guard\colon SO_i.State \rightarrow \textbf{Boolean}, T_0 \times \cdots \times T_n \rightarrow \textbf{void}\rangle \\
\textbf{guarded } \mathit{name} & : & \langle guard\colon \textbf{true}, T_0 \times \cdots \times T_n \rightarrow \textbf{void}\rangle \\
\textbf{guarded } \mathit{name} & : & \langle guard\colon SO_i.State \rightarrow \textbf{Boolean}, T_0 \times \cdots \times T_n \rightarrow T\rangle \\
\textbf{guarded } \mathit{name} & : & \langle guard\colon \textbf{true}, T_0 \times \cdots \times T_n \rightarrow T\rangle
\end{array}
$$

if $T \in$ **Class** and/or $T_i \in$ **Class** with $i \in \{0, \ldots n\}$.

Content of Memory Components $\forall Mem_i \in VSoC.ALS.\overline{Memory}$ the data type $Mem_i.Type$ needs to become serializable **if** $Mem_i.Type \in \{Object, ObjectArray\}$.

5.6.3.2 Software, Actor and Shared Object Behavioral-RT timing refinement

Software Socket $\forall a_i \in VSoC.ALS.\overline{Actor}$ where Actor a_i is mapped to a Software Socket $ss_i \in VSoC.VTA.\overline{SoftwareSocket}$, the type of $a_i.Behavior$ can be of kind *Active Sequential* and *Reactive Sequential*. All EET annotations inside $a_i.Behavior$ are mapped to the corresponding number of clock cycles of the reference frequency $f_{clk_{ref}}$ (used during timing estimation):

$$
\begin{array}{rcl}
M_{EET} & : & \mathbb{D} \rightarrow \mathbb{N}_{\geq 0} \\
M_{EET}(x) & = & |x| \cdot f_{clk_{ref}}
\end{array}
$$

with duration $\mathbb{D} = (value, unit)$ of the EET and the scaling function $|X|\colon unit \rightarrow \mathbb{R}^+$ (see Section 5.5.4.5).

Actor Socket $\forall a_i \in VSoC.ALS.\overline{Actor}$ where Actor a_i is mapped to an Actor Socket $as_i \in VSoC.VTA.\overline{HardwareSocket}|_{Socket_{HW}}$, the type of $a_i.Behavior$ can be of kind *Active Sequential*, *Reactive Sequential*, *Active Parallel* and *Reactive Parallel*. For the EET annotations inside $a_i.Behavior$ one of the following rules needs to be applied:

- mapping to the corresponding number of clock cycles of the reference frequency $f_{clk_{ref}}$ (as described for the Software Socket above) or
- manual insertion of `wait(N)` boundaries per EET block with the following properties:
  1. `N` $\in \mathbb{N}_{>0}$ is the number of clock cycles
  2. for each EET block E the total number of annotated clock cyles $\sum_{EET(E)} N \leq M_{EET}(E)$

Shared Object Socket $\forall so_i \in VSoC.ALS.\overline{Channel}$ where Shared Object so_i is mapped to a Shared Object Socket $sos_i \in VSoC.VTA.\overline{HardwareSocket}|_{Socket_{SO}}$, for the EET annotations inside the Shared Object's Services $so_i.Method$ and the Shared Object's Scheduler $so_i.Scheduler$, one of the following rules needs to be applied:

- mapping to the corresponding number of clock cycles of the reference frequency $f_{clk_{ref}}$ (as described for the Software Socket above) or
- manual insertion of `wait(N)` boundaries per EET block with the following properties:
  1. `N` $\in \mathbb{N}_{>0}$ is the number of clock cycles
  2. for each EET block E the total number of annotated clock cyles $\sum_{EET(E)} N \leq M_{EET}(E)$

5.6.3.3 RMI timing annotations

To represent the protocol timing overhead all RMI services need to be annotated with Estimated Execution Times (EET) or `wait(N)` statements to represent their corresponding execution times

Phase	$Socket_{SW}$ cycle annotation	$Socket_{HW}$ cycle annotation
initialization	variable[6]	static
look-ups:		
ObjectID to Address	static	static
ClientID to Address	static	static
ObjectID to Argument Address	static	static
ObjectID to Return Address	static	static
parameter serialization	variable[6]	N/A[7]
read serialized chunk	static	static
write serialized chunk	static	static
deserialize return value	variable[6]	N/A[7]
finalization	variable[6]	static

Table 5.7: Pre-defined RMI_Client_IF timing annotation points

on their mapped processing elements (i.e. Software or Hardware Socket). In particular, all services provided by the RMI interface need to be considered (for details see Definition 5.6.2.12):

RMI_Client_IF provides the following RMI client side services to be annotated:

1. Remote method call with no (`void`) return value

$$call_{procedure}\colon \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times Bitvector \rightarrow void$$
$$call_{procedure}(ClientID, ObjectID, MethodID, Parameters)$$

2. Remote method call with return value

$$call_{function}\colon \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times Bitvector \rightarrow Bitvector$$
$$Return_Value = call_{function}(ClientID, ObjectID, MethodID, Parameters)$$

The *RMI_Client_IF* can be accessed from an Actor mapped to a Software or Hardware Socket. In both cases the timing annotation of $call_{procedure}$ and $call_{function}$ represent the local computations for preparation, transfer and completion on the corresponding processing element. These times are RMI protocol implementation and target specific. Table 5.7 shows all pre-defined RMI_Client_IF timing annotation points which can be used to annotate the number of cycles for Actor to Hardware and Software Socket mappings.

The sequence of annotation phases for a client RMI call is:

$$\begin{aligned}
&\text{initialization} \rightarrow \\
&\text{ObjectID to Address} \rightarrow \text{ClientID to Address} \rightarrow \\
&\text{ObjectID to Argument Address} \rightarrow \text{ObjectID to Return Address} \rightarrow \\
&\text{parameter serialization} \rightarrow \\
&\text{read serialized chunk} \rightarrow \cdots \rightarrow \text{read serialized chunk} \rightarrow \\
&\text{write serialized chunk} \rightarrow \cdots \rightarrow \text{write serialized chunk} \rightarrow \\
&\text{deserialize return value} \rightarrow \\
&\text{finalization}
\end{aligned}$$

In these phases accesses to the OSSS-Channel (Bus or Point-to-Point) are using the `read_blocking` and `write_blocking` services of the `master_if`. Timing annotation (i.e.

[6] depends on the called method's total parameter and return value size
[7] data is represented serialized in all registers, no need to perfoem this conversation

	read	**write**	**member access**	**array access**
raw	static	static	N/A	N/A
simple	variable	variable	static	N/A
array	variable	variable	static	static

Table 5.8: Timing annotation points of Memory Accessor services

number of clock cycles) for these services are provided by the respective OSSS-Channel implementations (e.g. see Definition 5.6.2.15 and Definition 5.6.2.16 with the corresponding timing diagrams). In conjunction with the timing annotations of the RMI client interface services, as described above, communication times for RMI clients are completely covered.

RMI_Server_IF provides the following RMI server side services that can only be accessed from the Shared Object Socket implemented in hardware:

1. Waiting for requests

$$listen_for_action\colon void \to \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$$
$$[ClientID, ObjectID, MethodID] = listen_for_action()$$

   The execution of each call to `listen_for_action` takes *one cycle*.

2. Waiting for guard evaluation and scheduling

$$wait_for_guard\colon \mathbb{N}_{\geq 0} \times Boolean \to void$$
$$wait_for_guard(ClientID, is_busy)$$

   The execution of each call to `wait_for_guard` takes *one cycle*.

3. Reception of method parameters

$$receive_in_params\colon void \to Bitvector$$
$$Parameters = receive_in_params()$$

   The execution of each call to `receive_in_params` *depends on the duration (total number of cycles) of the parameter transfer from the corresponding client.*

4. Waiting for method execution to be finished

$$return_params_idle\colon \mathbb{N}_{\geq 0} \times Boolean \to void$$
$$return_params_idle(ClientID, is_busy)$$

   The execution of each call to `wait_for_guard` takes *one cycle*.

5. Provide method's return parameters

$$provide_return_params\colon \mathbb{N}_{\geq 0} \times Bitvector \to void$$
$$provide_return_params(ClientID, Return_Value)$$

   The execution of each call to `provide_return_params` *depends on the duration (total number of cycles) of the return value reception of the corresponding client.*

5.6.3.4 Memory timing annotations

For adding memory `read` and `write` access times to VTA memory components (see Definition 5.6.2.6) the provided interface functions of Memory Accessors (see Definition 5.6.2.7) can be annotated with delay cycles as shown in Table 5.8.

[8] number of total cycle depend on the ratio of accessed data type size in bit and the data with $DWidth$ of the Memory Accessor

In the raw read, write and read/write interfaces the memory access delay is constant for all accesses. In the simple read, write and read/write interfaces the memory access delay of all basic read accesses is constant, as well as for all basic write accesses. But anyhow, the number of total delay cycle for each individual access depends on the ratio of accessed data type size (in bit) and the data with $DWidth$ of the Memory Accessor:

$$\text{access delay}(T) = \left\lceil \frac{size_of(T)}{DWidth} \right\rceil$$

Delays of member accesses in simple and array access interfaces are constant. Also delays of array accesses are constant. A concatenation of member and/or array accesses results in an addition of the access delays of each indirection.

5.6.4 Operational Semantics

Like in Section 5.5.6 the operational semantics of the Virtual Target Architecture Layer is expressed using timed automata (TA) in Uppaal. The timed automata representation expresses the communication protocol refinement using the RMI- and OSSS-Channel modeling elements including the presented timing annotations.

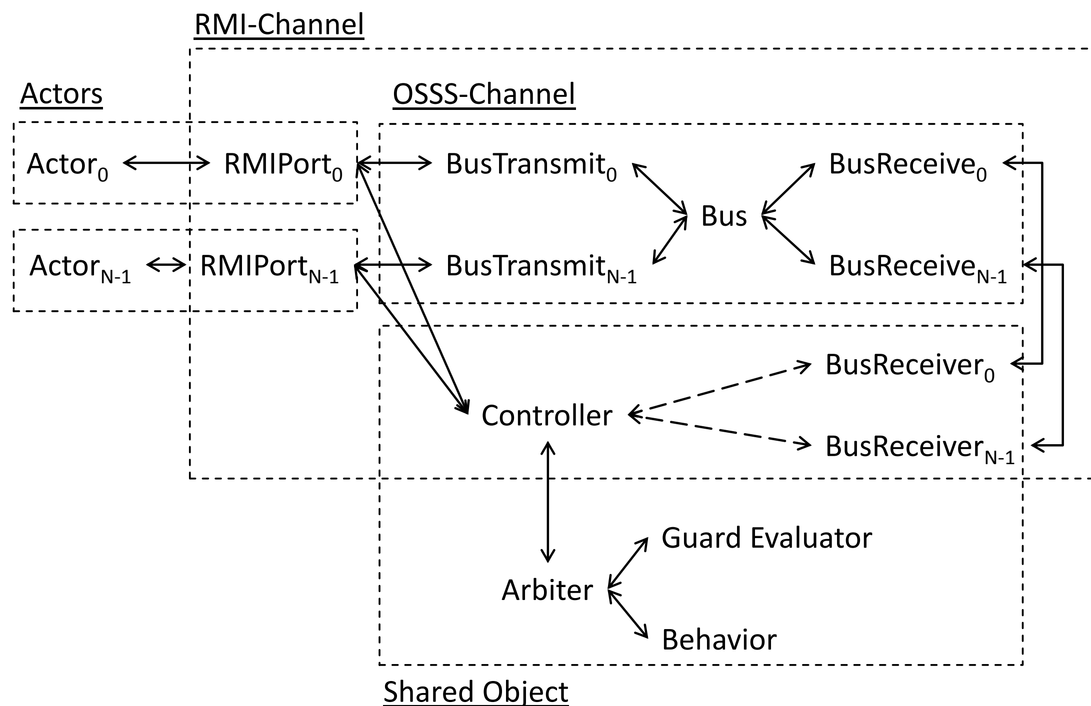


Figure 5.67: Overview of VTA Shared Object, RMI- and OSSS-Channel split-up

Figure 5.67 provides an overview of the network of timed automata modeling an RMI- and OSSS-Channel implementing a shared bus. It is the same example as used in Section 5.5.6 and consists of two actors, one putting data into a shared FIFO and another one getting data from this FIFO. On the Virtual Target Architecture Layer both actors are connected to the FIFO Shared Object through (an arbitrated) shared bus. Compared to the Application Layer model, only the automata inside the RMI-Channel box have been modified or newly added to the TA network. The Actor (see Section 5.5.6.1) and the Shared Object Arbiter, Guard Evaluator and Behvior TAs (see Section 5.5.6.3) have not been modified. In the following subsections only the modified and newly added TAs will be described.

5.6.4.1 RMI Port

The RMI Port (see Figure 5.68) replaces the Port TA of the Application Layer (see Section 5.5.6.2). The RMI Port describes the RMI protocol and the timing of the different phases according to Table 5.7. The protocol works along the following phases:

idle waits until the service call gets requested by the connected Actor through the `call?` event.

C0_initialize models the initialization phase of the RMI call and waits for a configurable time interval $[initialize[BCET], initialize[WCET]]$.

C1_lookup models different ID to (bus) address look-ups, e.g. ObjectID and ClientID to address, ObjectID to argument address and ObjectID to return address. It waits for a configurable time interval $[lookup[BCET], lookup[WCET]]$.

C2_arg_serialisation models the duration of the service argument/parameter serialization. The model uses the delay *serialize_base* for the serialization of `BIT_WIDTH`[9] elements and calculates the delay interval as:

$$[serialize_base[BCET] \cdot calculate_cycles(method_argument_size[ARG], BIT_WIDTH),$$
$$serialize_base[WCET] \cdot calculate_cycles(method_argument_size[ARG], BIT_WIDTH)]$$

The `calculate_cycles` function is defined in Listing 5.20.

C3_ID_transmission writes the MethodID to the corresponding address (identifying the client number). The ID is transferred using a simple (non-burst) bus write access. The `bus_write_single` (see Listing 5.20) models a simple write transfer on the bus. The `sr[conn-1]!` event is used to send a request to the Bus Master/Transmit TA. The `srr[conn-1]?` event indicates send request return (i.e. the completion of the bus access). The duration of the bus access depends on the delay until the request gets granted by the bus arbiter and the duration of the bus write access. These delays are defined in the Transmit and Bus TAs.

C4_waiting_SO_access polls the shared bus for Shared Object access grant of the requested service. The polling is modeled through a single (non-burst) read transfer using the `bus_read_single` function.

C5_stream_args writes the serialized service arguments to the Shared Object Socket's argument memory. The duration of the burst access is modeled by Bus Transmit and Bus TA. The function `bus_write_burst` (see Listing 5.20) passes the start address, data and length of the burst to the timing model of the bus. The length of the burst transfer is calculated using the `calculate_cycles` function.

C6_wait_completion polls the shared bus for Shared Object service completion. The polling is modeled through a single (non-burst) read transfer using the `bus_read_single` function.

C7_stream_results reads the serialized service return value from the Shared Object Socket's return value memory. The function `bus_read_burst` (see Listing 5.20) passes the start address and length of the burst to the timing model of the bus. The length of the burst transfer is calculated using the `calculate_cycles` function.

C8_deserialise_results models the duration of the service return value deserialization. Like for the argument serialization delay interval is defined and calculated as:

$$[serialize_base[BCET] \cdot$$
$$calculate_cycles(method_argument_size[RET], BIT_WIDTH),$$
$$serialize_base[WCET] \cdot$$
$$calculate_cycles(method_argument_size[RET], BIT_WIDTH)]$$

C9_finalize models the finalization phase of the RMI call, waits for a configurable time interval $[finalize[BCET], finalize[WCET]]$ and issues the `ret!` event to indicate the completion of the service call at the Actor TA.

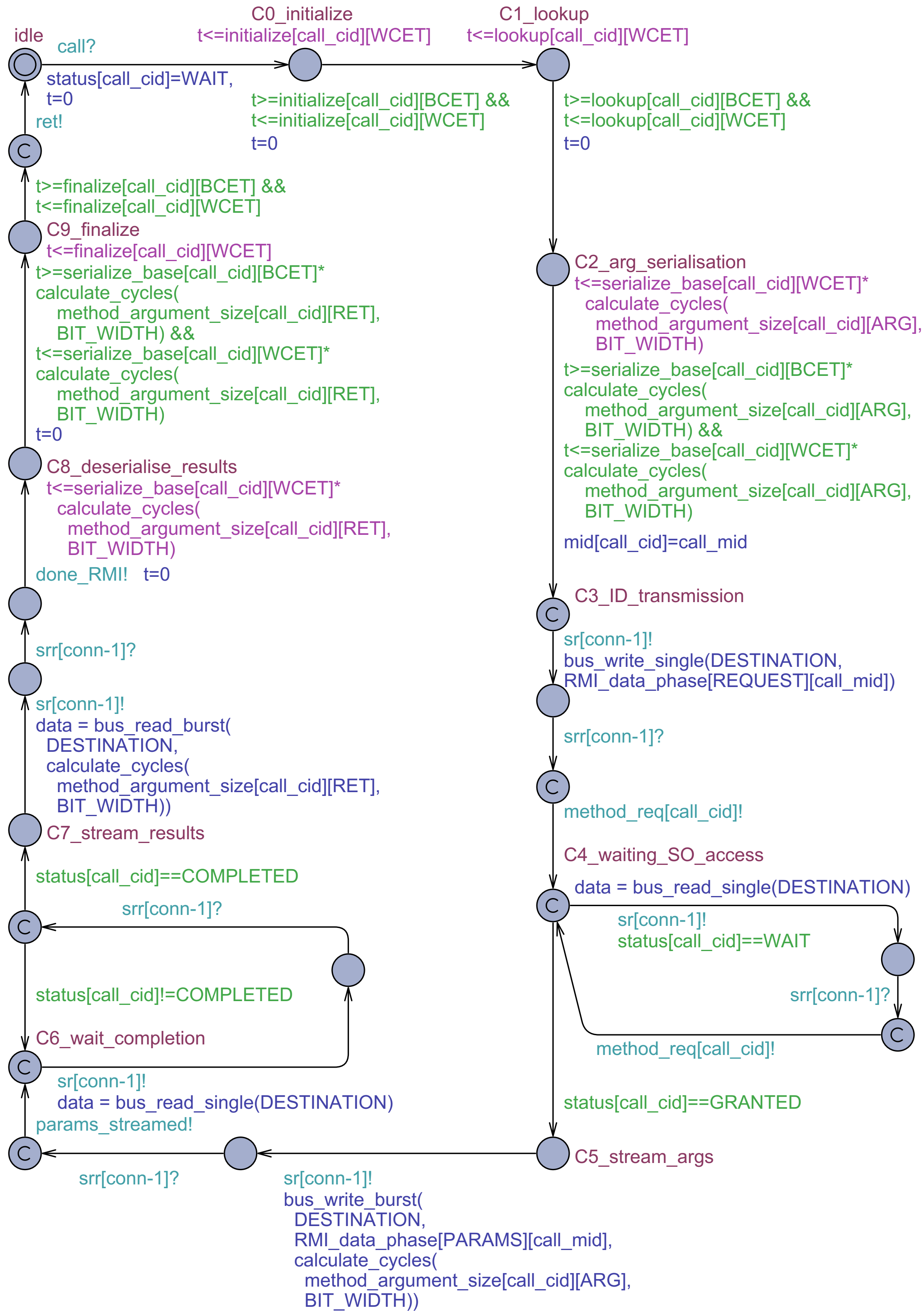


Figure 5.68: RMI Port modeled as timed automaton

```
1  clock t;
2  int data;
3
4  int calculate_cycles(int data_size, int bit_width) {
5     int num_cycles ← 0;
6     if (data_size ≤ bit_width)
7       num_cycles ← 1;
8     else {
9        num_cycles ← data_size / bit_width;
10       if (data_size % bit_width != 0)
11          num_cycles ← num_cycles + 1;
12    }
13    return num_cycles;
14 }
15
16 void bus_write_single(int addr, int data) {
17    rnw ← false;
18    address ← addr;
19    burst ← false;
20    wire ← data;
21 }
22
23 int bus_read_single(int addr) {
24    rnw ← true;
25    address ← addr;
26    burst ← false;
27    return wire;
28 }
29
30 void bus_write_burst(int addr, int data, int size) {
31    rnw ← false;
32    address ← addr;
33    burst ← true;
34    burst_length ← size;
35    wire ← data;
36 }
37
38 int bus_read_burst(int addr, int size) {
39    rnw ← true;
40    address ← addr;
41    burst ← true;
42    burst_length ← size;
43    return wire;
44 }
```

Listing 5.20: RMI Port functions

5.6.4.2 Shared Object Socket

For the Shared Object only a very few modifications, compared to the Application Layer TAs, have been performed. The Shared Object Controller (see Figure 5.69) is now synchronized with the RMI Port after the arguments of the requested service have been completely streamed to the Shared Object's argument memory using the `params_streamed` event. After completion of the RMI protocol the Shared Object Controller and the RMI Port are synchronized using the `done_RMI` event.

All other TAs of the Shared Object Socket (Arbiter, Guard Evaluator and Behavior) are the same as introduced in the operational semantics section of the Application Layer (see Section 5.5.6).

5.6.4.3 OSSS-Channel

Only a description of a generic shared bus model will be described in the following sections. A point-to-point channel implementation is not described here, because it can be modeled as a simple data width dependent delay.

[9]in our model this is the bit width of the shared bus

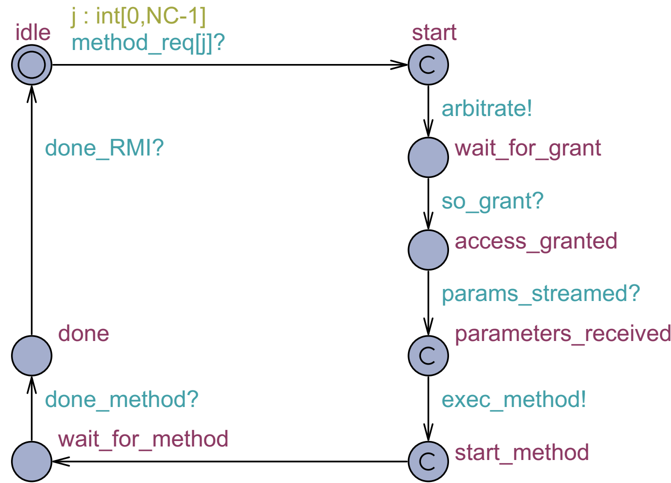


Figure 5.69: Shared Object Controller

Bus Master/Transmitter Figure 5.70 shows the TA template of the master bus interface (called transmitter). It synchronizes with the send request `sr` event, issued by the RMI Port. Before accessing the shared bus, the permission of the bus arbiter is requested using the `busreq` event. The `grant` event issued by the bus arbiter grant access permission to the shared bus. The `send` event is used to synchronize with the shared bus model. The `receive` event models the release of the bus. Finally the send request receive `srr` event synchronizes with the RMI Port and notifies the completion of the requested bus access.

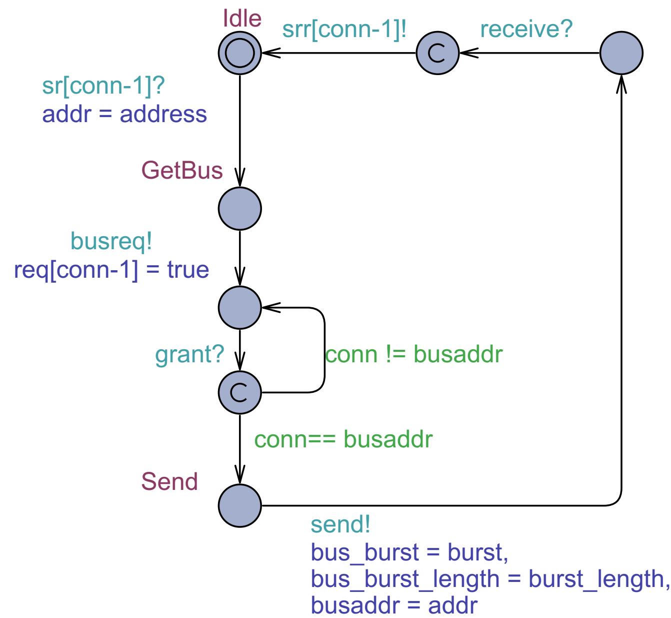


Figure 5.70: Bus master interface (transmitter) modeled as timed automaton

Bus Arbiter The shared bus arbiter, shown in Figure 5.71, has the same structure and implements the same scheduling algorithms as the Shared Object arbiter. The `busreq` event starts the selected scheduling algorithm (for the different scheduling algorithms see Section B.1). The `grant` event notifies the completion of the scheduling algorithm. The `busaddr` shared variable is used to indicate the ID of the granted master. Upon completion of a bus transfer (notified through the `receive` broadcast event) an new scheduling phase is initiated.

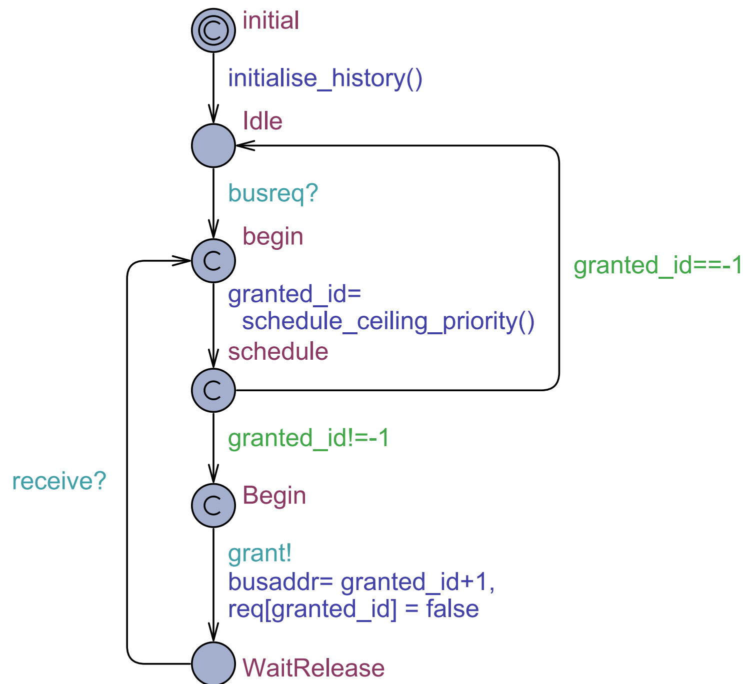


Figure 5.71: Bus arbiter modeled as timed automaton

Bus Figure 5.72 shows the TA model of the shared bus medium. The `send` event initiates the bus access. A single transaction has the duration $D1$ and a burst transaction has the duration $D1 + D2 * (bus_burst_length - 1)$. The `receive` broadcast event notifies the completion of the shared medium access.

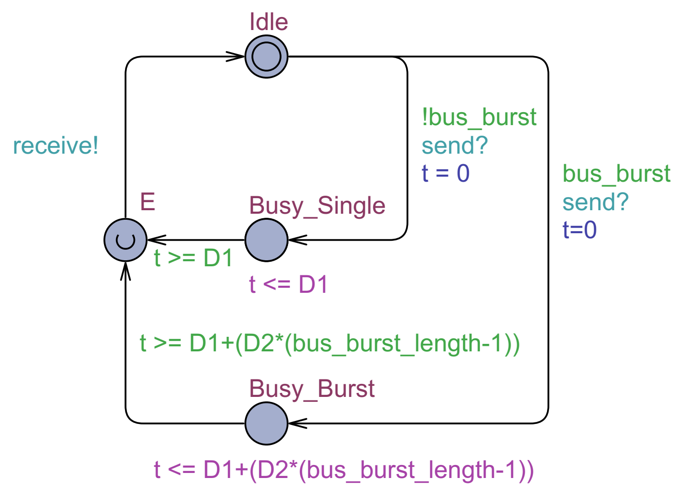
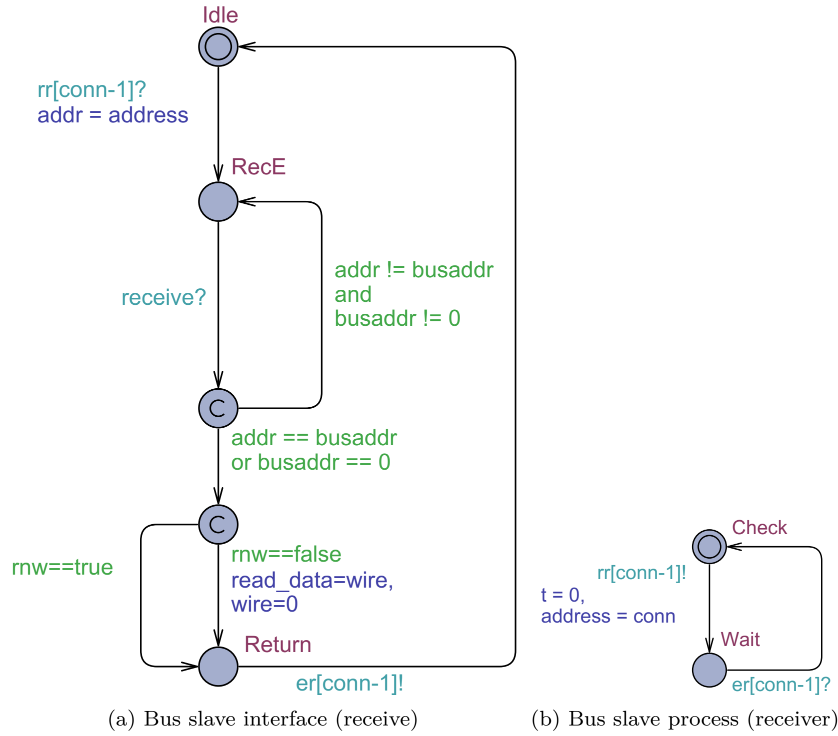


Figure 5.72: Shared bus medium modeled as timed automaton

Bus Slave/Receiver Figure 5.73a models the bus slave interface (called receive). The automaton gets started by the receive request `rr` event. Upon `receive` broadcast event synchronization, the slave interface is activated at the end of a transaction on the shared bus. After reading or writing (`rnw` is the "read not write" flag) data from the shared medium `wire` the end receive `er` event is notified.

Figure 5.73 is a simple TA that implements a bus slave process that listens on the shared medium continuously. The receive request `rr` event triggers the slave interface automaton and the end receive `er` event synchronizes upon completion of the slave interface's reception.



(a) Bus slave interface (receive) (b) Bus slave process (receiver)

Figure 5.73: Bus slave timed automaton models

For keeping the TA model as simple as possible, only the master interface bus access (including contention and bus arbitration) has been modeled. The transfer of data payload (e.g. IDs, service argument and return value vectors) is not part of this model. For this reason the bus slave interface and bus slave process TAs could also be omitted here. Nevertheless, for symmetry both interfaces of the shared bus have been included in the presented TA model.

5.6.4.4 Putting it all together

Listing B.5 instantiates the described TA templates of the Virtual Target Architecture model of the bounded FIFO producer-consumer example. In line 4-15 of Listing 5.21 the EET annotations from the Application Layer model are defined:

$$
\begin{aligned}
Put.EET_0 &:= \texttt{put_EET_before} = [10, 15] \\
Put.EET_1 &:= \texttt{put_EET_after} = [4, 5] \\
Get.EET_0 &:= \texttt{get_EET_before} = [10, 15] \\
Get.EET_1 &:= \texttt{get_EET_after} = [4, 5] \\
FIFO.T_{GE} &:= \texttt{EET_eval} = [1, 1] \leq NC \cdot \max\left(FIFO.EET_{g0}, FIFO.EET_{g1}\right) \\
FIFO.EET_{sched} &:= \texttt{EET_sched} = [1, 1] \\
FIFO.EET_{put} &:= \texttt{EET_service[0]} = [2, 5] \\
FIFO.EET_{get} &:= \texttt{EET_service[1]} = [2, 5]
\end{aligned}
$$

In line 20-24 of Listing 5.21 the EET RMI annotations of the Virtual Target Layer model are defined:

$$
\begin{aligned}
Put.RMIPort.init &:= \texttt{rmi_port_init[0]} = [1, 1] \\
Get.RMIPort.init &:= \texttt{rmi_port_init[1]} = [1, 1] \\
Put.RMIPort.lookup &:= \texttt{rmi_port_lookup[0]} = [1, 2] \\
Get.RMIPort.lookup &:= \texttt{rmi_port_lookup[1]} = [1, 2] \\
Put.RMIPort.serialize_base &:= \texttt{rmi_port_serialize_base[0]} = [2, 3]
\end{aligned}
$$

$$Get.RMIPort.serialize_base := \texttt{rmi_port_serialize_base[1]} = [2,3]$$
$$Put.RMIPort.deserialize_base := \texttt{rmi_port_deserialize_base[0]} = [1,2]$$
$$Get.RMIPort.deserialize_base := \texttt{rmi_port_deserialize_base[1]} = [1,2]$$
$$Put.RMIPort.final := \texttt{rmi_port_final[0]} = [1,1]$$
$$Get.RMIPort.final := \texttt{rmi_port_final[1]} = [1,1]$$

RET upper bounds are defined in line 32+33:

$Put.RET := \texttt{PUT_PERIOD} = 110$
$Get.RET := \texttt{GET_PERIOD} = 110$

```
1  // Application model timing annotations:
2
3  // Timing annotations, written as [BCET, WCET] intervals
4  int put_EET_before[2] ← {10, 15};
5  int put_EET_after[2]  ← {4, 5};
6
7  int get_EET_before[2] ← {10, 15};
8  int get_EET_after[2]  ← {4, 5};
9
10 int EET_eval[2] ← {1, 1};
11 int EET_sched[2] ← {1, 1};
12
13 int[0,FIFO_SIZE] num_elements ← 0;
14 // Shared Object service execution time [PUT, GET][BCET, WCET]
15 int EET_service[2][2] ← {{2, 5}, {2, 5}};
16
17 // Virtual Target Architecture timing annotations:
18
19 // RMI initiator execution times [client0, client1][BCET, WCET]
20 int rmi_port_init[2][2] ← { {1,1}, {1,1} };
21 int rmi_port_lookup[2][2] ← { {1,2}, {1,2} };
22 int rmi_port_serialize_base[2][2] ← { {2,3}, {2,3} };
23 int rmi_port_deserialize_base[2][2] ← { {1,2}, {1,2} };
24 int rmi_port_final[2][2] ← { {1,1}, {1,1} };
25
26 // A simple bus transmission takes D1 time units
27 const int D1 ← 2;
28 // A bus burst transaction takes D1+(D2*(burst_length-1)) time units
29 const int D2 ← 1;
30
31 // timing requirements for put and get clients
32 const int PUT_PERIOD ← 110;
33 const int GET_PERIOD ← 110;
```

Listing 5.21: Timing annotations

```
1  method_type mid_request[NC];
2  method_type mid_request_guarded[NC];
3  method_type so_mid ← NOP;
4  status_type call_status[NC];
5
6  urgent chan put_call, put_ret;
7  client_type put_cid;
8  method_type put_call_mid;
9
10 Put ← Actor(0, PUT, put_cid, put_call_mid, put_call, put_ret,
11             put_EET_before, put_EET_after);
12
13 // to RMI Socket
14 broadcast chan method_req[NC];
15 urgent chan params_streamed, done_RMI;
16
17 Put_Port ← RMIPort(put_cid, put_call_mid, put_call, put_ret,
18                    method_req, params_streamed, done_RMI,
19                    mid_request, call_status,
20                    rmi_port_init,
```

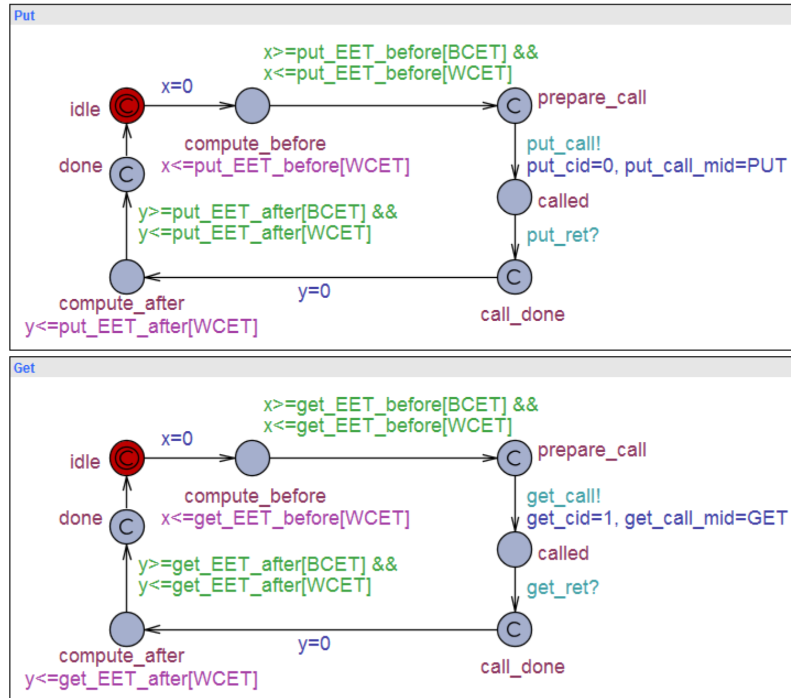


Figure 5.74: Put and Get Actors

```
21                      rmi_port_lookup,
22                      rmi_port_serialize_base,
23                      rmi_port_deserialize_base,
24                      rmi_port_final,
25                      1, 1, 32);
26
27  urgent chan get_call, get_ret;
28  client_type get_cid;
29  method_type get_call_mid;
30
31  Get ← Actor(1, GET, get_cid, get_call_mid, get_call, get_ret,
32              get_EET_before, get_EET_after);
33
34  Get_Port ← RMIPort(get_cid, get_call_mid, get_call, get_ret,
35                      method_req, params_streamed, done_RMI,
36                      mid_request, call_status,
37                      rmi_port_init,
38                      rmi_port_lookup,
39                      rmi_port_serialize_base,
40                      rmi_port_deserialize_base,
41                      rmi_port_final,
42                      2, 1, 32);
43
44  // to Arbiter
45  urgent broadcast chan arbitrate;
46  urgent chan so_grant, exec_method, done_method;
47  method_type scheduled_mid;
48
49  // to Receiver
50  int read_data, write_data;
51
52  SO_Ctrl ← SO_Controller(method_req,
53                           arbitrate, so_grant, params_streamed, exec_method,
        granted_mid,
54                           done_method, done_RMI,
55                           read_data, write_data);
56
57  // Arbiter to Shared
58  urgent chan call_so, ret_so;
```

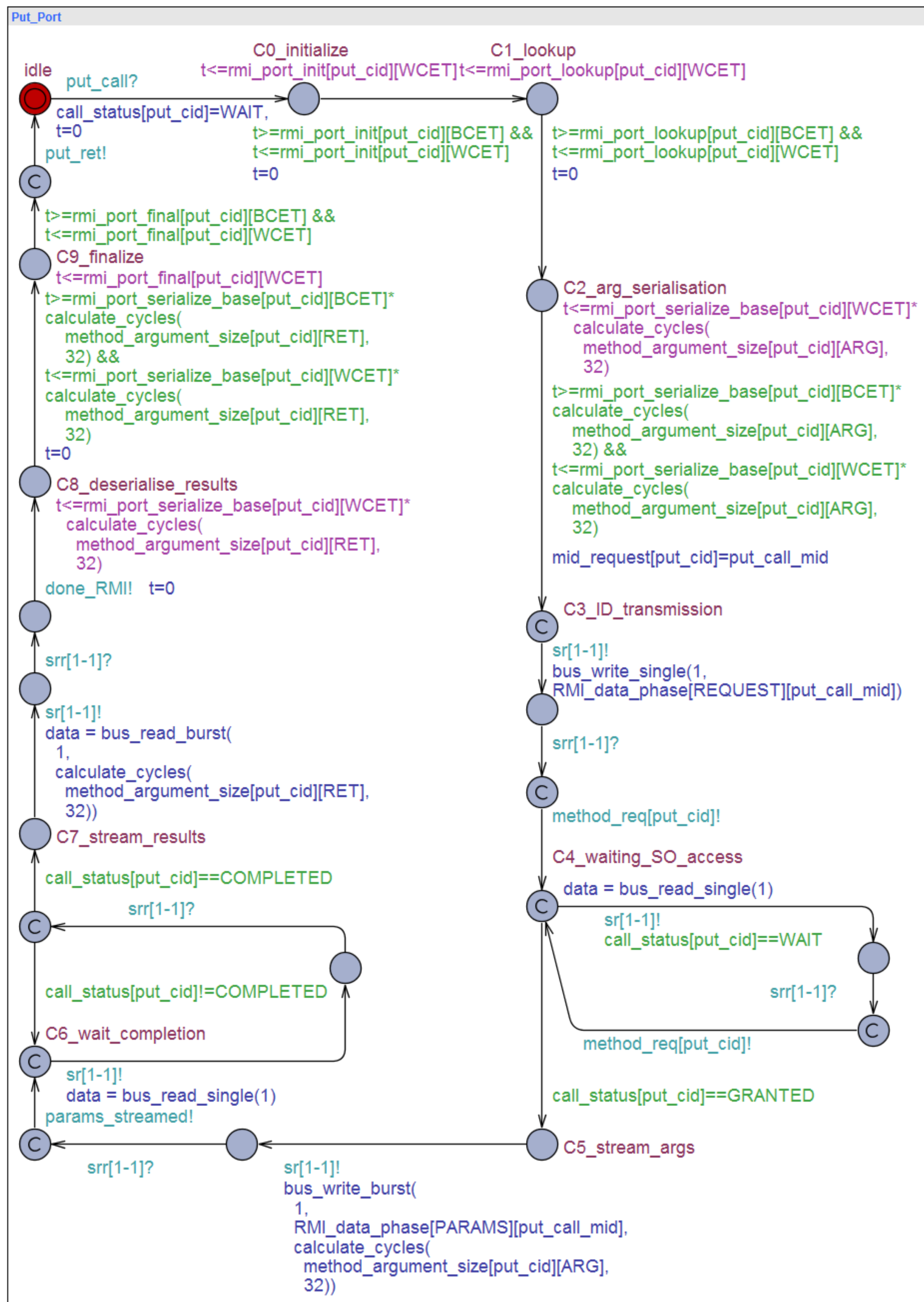


Figure 5.75: RMI Port of Put Actor

Figure 5.76: RMI Port of Get Actor

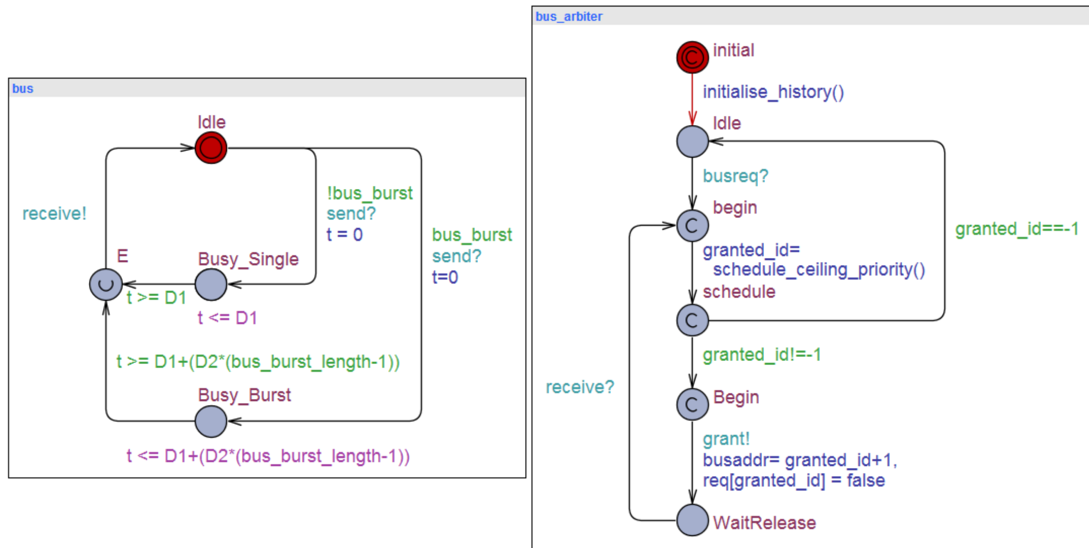


Figure 5.77: Bus and Bus Arbiter

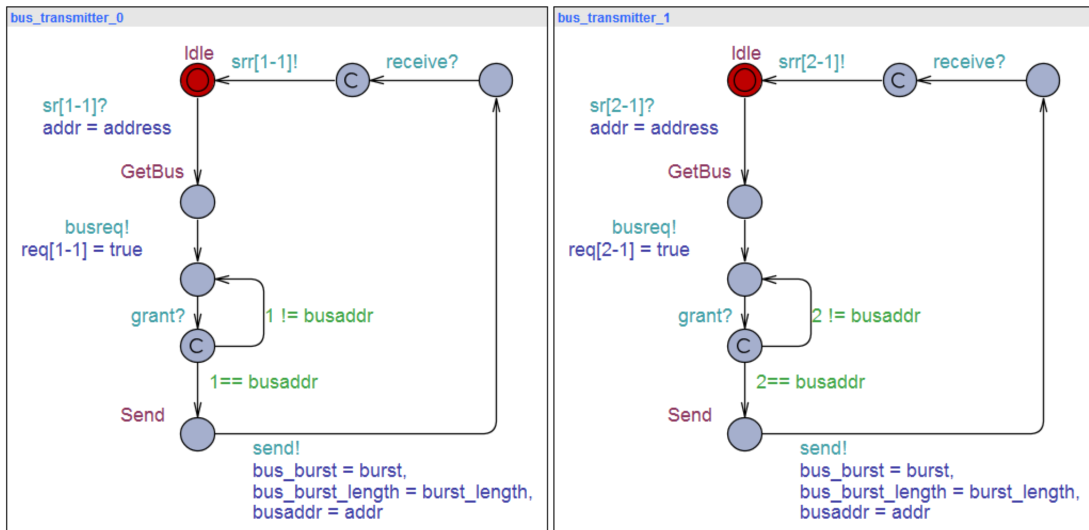


Figure 5.78: Bus Masters/Transmitters of Put and Get Actors

```
59
60  // Arbiter to Guard Evaluator
61  urgent chan eval_guards, eval_guards_done;
62
63  SO_GE ← SO_Guard_Evaluator(eval_guards, eval_guards_done,
64                            mid_request, mid_request_guarded,
65                            num_elements,
66                            EET_eval);
67
68  SO_Arb ← SO_Arbiter(arbitrate,
69                      eval_guards, eval_guards_done,
70                      so_grant, exec_method,
71                      mid_request, mid_request_guarded, call_status,
72                      done_method,
73                      call_so, granted_mid, scheduled_mid, ret_so,
74                      EET_sched);
75
76  SO_Beh ← SO_Behavior(call_so, ret_so, scheduled_mid,
77                       EET_service, num_elements);
78
```

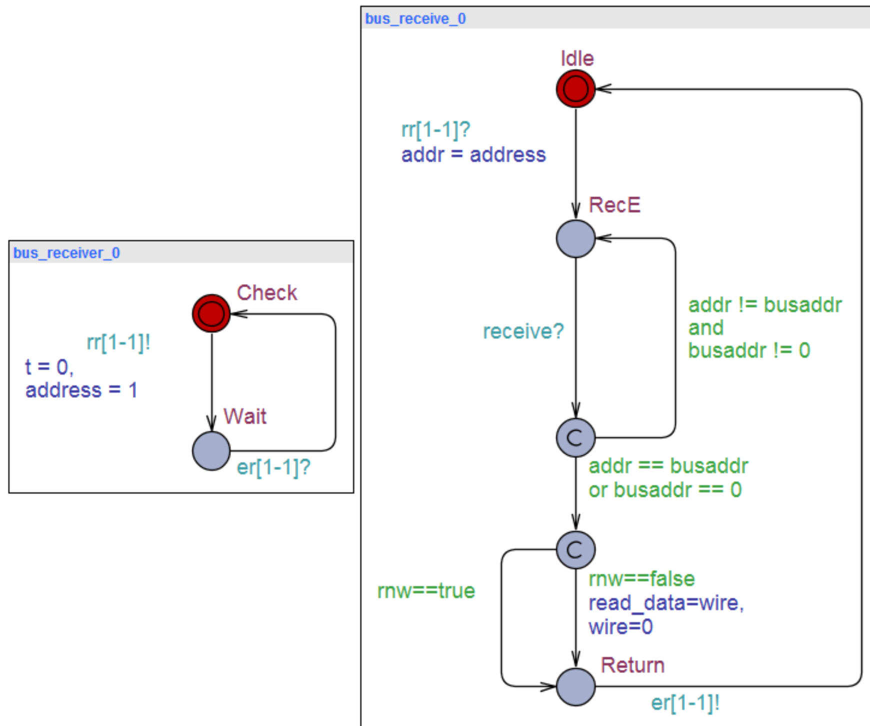


Figure 5.79: Bus Slave/Receiver of Shared Object Socket

```
79  bus ← Bus(D1, D2);
80  bus_transmitter_0 ← Bus_Transmit(1);
81  bus_transmitter_1 ← Bus_Transmit(2);
82  bus_receiver_0 ← Bus_Receiver(1);
83  bus_receive_0 ← Bus_Receive(1, read_data, write_data);
84  bus_arbiter ← Bus_Arbiter();
85
86  system Put, Put_Port,
87         Get, Get_Port,
88         SO_Ctrl, SO_GE, SO_Arb, SO_Beh,
89         bus_transmitter_0, bus_transmitter_1,
90         bus_receiver_0, bus_receive_0,
91         bus, bus_arbiter;
```

Listing 5.22: System definition

5.6.4.5 Properties

Like in Section 5.5.6.5 a simplified version of TCTL (Timed Computation Tree Logic)[10] is used to check whether all properties of the Application Layer model are still fulfilled in the Virtual Target Architecture model.

Safety properties For the FIFO Shared Object Virtual Target Architecture model in Listing B.5 we can proof that the following safety properties are fulfilled:

- $A\Box$ (*not deadlock*)

  The system does never deadlock.
- $A\Box$ ($\texttt{Put.done} \Rightarrow \texttt{Put.x} \leq \texttt{PUT_PERIOD}$)

  The completion of a `put` call takes never longer than `PUT_PERIOD`.
- $A\Box$ ($\texttt{Get.done} \Rightarrow \texttt{Get.x} \leq \texttt{GET_PERIOD}$)

  The completion of a `get` call takes never longer than `GET_PERIOD`

[10] More information on the used query language can be found in Appendix 3.3.4.

- $A\Box$ ((`Put.call_done` $\Rightarrow$ *not* `Get.call_done`) *and* (`Get.call_done` $\Rightarrow$ *not* `Put.call_done`))

  The `put` and `get` service call executions are always mutual exclusive.
- $A\Box$ (`num_elements` $\leq$ `FIFO_LIMIT`)

  The total number of elements in the Shared Object's buffer is always less or equal to `FIFO_LIMIT`. Given the timing annotation from Listing 5.21 `FIFO_LIMIT` depends on the following parameters:

priority	scheduling algorithm	FIFO_LIMIT
Put > Get	static priority	FIFO_SIZE:=5
Put < Get	static priority	1
-	ceiling priority	1
-	round robin	1
-	modified round robin	1

Liveness properties For the FIFO Shared Object Virtual Target Architecture model in Listing B.5 we can proof that the following liveness properties are fulfilled:

- `Get.called` $\leadsto$ `Get.call_done`

  When a `get` call is requested it will be eventually served by the Shared Object.
- `Put.called` $\leadsto$ `Put.call_done`

  When a `put` call is requested it will be eventually served by the Shared Object.
- `Put.call_done` $\leadsto$ `Get.call_done`

  When a `put` call is completed a `get` call eventually completes.

5.7 Target Platform[11]

5.7.1 Introduction

The *Target Platform* represents the physical implementation of a SoC as defined in Chapter 2. To end up with an implementation on a *Target Platform* the following steps have to be performed, as described in Section 5.2:

1. the application, using the modeling elements of the Application Layer, described in Section 5.5.2, has to be mapped on the *Virtual Target Architecture Layer*, as described Section 5.6,
2. the application mapped on the virtual target architecture has to be transformed or synthesized to a proper representation for further processing (see Chapter 7), and
3. vendor specific compilers and synthesis tools have to be used to implement the design on the chosen target platform (see Section 7.9).

While the Virtual Target Architecture Layer model representation can be modified and retargeted to represent different target platforms, the Target Platform representation ties the VTA model to a fixed implementation. As already described in Chapter 2, currently only Xilinx platform FPGAs are supported. More details about the supported Xilinx platforms can be found in Appendix E.

5.7.2 Modeling Elements

Figure 5.80 shows the supported Xilinx Target Platform modeling elements, including their relationship to the Virtual Target Architecture modeling elements presented in Section 5.6. The following subsections only briefly introduce the Xilinx Target Platform elements. More details can be found in Section 7.4.

[11] This section is based on [44].

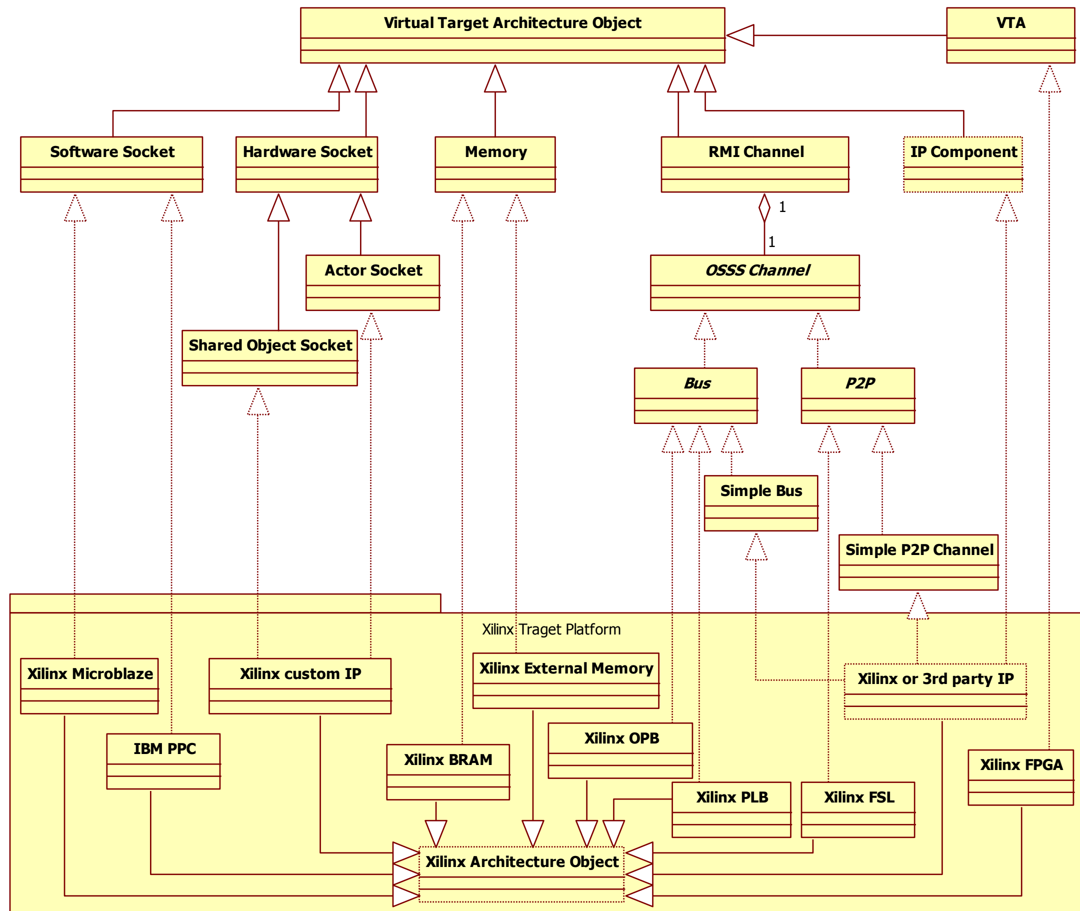


Figure 5.80: Virtual Target Architecture Meta Model with Xilinx FPGA Target Platform elements

5.7.2.1 Software Processing

The software processing elements replace the *Software Socket* in the Virtual Target Architecture Layer model. Supported software processing elements are:

Xilinx MicroBlaze configurable soft-core processor [103] and

IBM PPC PowerPC 405 RISC processor [26].

Both software processors include the following configurable components:

- local and exclusive data and instruction memory
- timer with optional interrupt
- bus burst interface (optional)
- external data and instruction memory using exclusive memory bus (optional)
- UART with or without interrupt (optional)
- debug module (optional)

More details about the supported software processor configuration can be found in Section 7.4.1.

Actors mapped to these software processing elements are cross-compiled for the respective micro-architecture and linked with start-up and RMI communication libraries. For more details on software synthesis see Section 7.6.

5.7.2.2 Hardware Processing

The hardware processing element replaces the *Hardware Socket* and its derived *Actor Socket* and *Shared Object Socket*. Both elements describe custom hardware whose behavior is defined at the Behavior and the Application Layer. The custom defined Actors and Shared Objects mapped to the respective Sockets are transformed into a synthesisable VHDL representation and integrated as **Xilinx custom IP** components (more details about the custom hardware synthesis can be found in Section 7.7). Depending on the ports, interfaces and bindings as defined on the Virtual Target Architecture Layer, the Xilinx custom IP component can have multiple of the following interfaces:

Signal: For the connection with Xilinx or other 3rd party IP component's signal interfaces (including the Simple Bus and Simple P2P Channel Section 5.6.2.9), memory components and connection with SoC/FPGA pins.

IPIF: For the connection with Xilinx OPB [105] and Xilinx PLB [25] communication IP components. The Xilinx IP Interface (IPIF) [108, 112] is a description that can be automatically translated into VHDL/Verilog modules that act as the interface between user logic and the PLB or OPB buses. It handles address range checking, implements user-defined registers, supports fixed length burst transfers including read or write FIFOs. For more details on the usage of IFIP during interface synthesis see Section 7.8.

FSL: For the connection with Xilinx Fast Simplex Link (FSL) [24] communication. The FSL is a unidirectional point-to-point communication channel using an unshared non-arbitrated communication mechanism supporting control and data communication. It has a configurable data size and FIFO size for buffered message passing. More details about using the FSL interface can be found in Section 7.6.

5.7.2.3 Memory

To represent dedicated memory elements from the Virtual Target Architecture Layer FPGA internal Block-RAM (BRAM) [18] and external memory is supported. The synthesis of complex memory configurations including user-defined object access is not in the scope of this work. The support of dedicated memory is restricted to the following usage:

Xilinx BRAM: can be configured with different data and address widths in single- or dual-ported mode using the following access modes:

- raw signal access
- Xilinx OPB or PLB slave

Xilinx External Memory: can be configured with different data and address widths in single- or multi-ported mode using the following access modes:

- Xilinx OPB [87] or PLB slave [89]
- Xilinx Multi-Channel (MCH) Interfaces (up to 4) [19]

For more details and constraints on memory component usage for platform synthesis see Section 7.4.3.

5.7.2.4 Communication

In addition to the *Simple Bus* and *Simple P2P Channel* (see Section 5.6.2.9) the following communication channel IP components are supported for RMI:

- Xilinx OPB [105] (shared bus)
- Xilinx PLB [25] (shared bus)
- Xilinx FSL [24] (unidirectional point-to-point)

For more details see Section 7.4.4.

5.7.2.5 IP

Other IP components using a signal level interface can be included using the **Xilinx of 3rd party IP** component element. The *Simple Bus* and *Simple P2P Channel* are just two examples. Another example is a specific actuator or sensor interface. But also the signal level usage of a memory (also a Xilinx BRAM) can be modeled as custom IP component, i.e. when using the BRAM as a FIFO or line buffer.

5.7.2.6 SoC

The **Xilinx FPGA** component specifies the boundary of the chip, including all external ports and mappings to the pins of the FPGA. For supported FPGAs see Appendix E.

5.8 Summary

In this summary a comparison between the simulation independent goals of Chapter 2 and the modeling elements, as defined in this chapter. is provided. Table 5.9 presents the goals and a brief discussion regarding their fulfillment by the presented methodology and modeling elements. In summary, the methodology fulfills all implementation independent modeling goals (M5-M9, M12, M14), except for the support of modeling multitasking (M11) and operating systems (M13). These are considered as future work. Extensions for modeling of multitasking and operating systems have already been described in [48, 23, 17].

The other simulation model dependent modeling goals (M1-M4, M10) will be discussed in the summary of the simulation chapter (see Section 6.7), as well as the analysis goals (A1-A5). The synthesis goals (S1-S8) will be reviewed and discusses in the summary of the synthesis chapter (see Section 7.10).

Table 5.9: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled)

ID	Goal	Status	Comment
G2	Introduce a notion of time for the SW parts	●	Timing annotations presented in Section 5.5.4 are called Estimated Execution Times (EET) enable timing annotations of Actors. Required Execution Time (RET) annotations enable to check timing requirements within Actors. Actors mapped on Software Sockets (see Section 5.6.3) of the Virtual Target Architecture represent the SW part (Software Tasks) of the SoC platform.
M5	To be able to cover untimed (purely functional) models, transaction-level models and cycle accurate models	●	The Behavioural Layer (see Section 5.4) enables untimed (purely functional) system modeling. The Application Layer (see Section 5.5) enables transaction-level modeling, because communication between Actors and Shared Objects is performed by abstract service calls. Application Layer models mapped to Virtual Target Architecture Layer models (see Section 5.6) enables cycle accurate system modeling.

continued on next page

Table 5.9: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
M6	Methodology needs to provide modeling elements which allow to describe the communication	●	At the Behavioural Layer pre-defined channels for the communication between Behaviors are provided (see Section 5.4.2.3). At the Application Layer, Shared Objects are provided for the communication between Actors (see Section 5.5.2.3). At the Virtual Target Architecture Layer OSSS-RMI Channel (see Section 5.6.2.8) containers and OSSS-Channels (see Section 5.6.2.9) are provided for modeling communication at the SoC platform.
M7	Easy HW/SW repartitioning of the design (a SW module can be replaced by a HW module without manually modifying its communication interfaces)	●	At the Application Layer, Actors are used to model HW and SW components. They have the same connections (using the same port to interface binding concept) to Shared Objects. When replacing an Actor modeling a Software Task with an Actor modeling a Hardware Module no modification on the other components (incl. Shared Objects) of the Application Layer become necessary.
M8	Provide constructs for a uniform interface description	●	At the Behavioural Layer, channels with pre-defined interface services are provided. At the Application Layer, Shared Objects with user-defined service interfaces are provided. During mapping of the Application to the Virtual Target Architecture the Shared Object service interfaces are preserved.
M9	Provide modeling constructs for abstract communication	●	At the Behavioural Layer pre-defined channels for the communication between Behaviors are provided (see Section 5.4.2.3). At the Application Layer, Shared Objects are provided for the communication between Actors (see Section 5.5.2.3). Both communication concepts abstract from the signal level implementation of the communication. At the Virtual Target Architecture Layer service calls on Shared Objects are implemented by RMI-Channels and OSSS-Channels.
M11	Support of multitasking	○	Not supported in this work. Extensions of OSSS for software multitasking can be found in [48, 23, 17].
M12	Consideration of (real-)time constraints	●	The combination of Estimated Execution Times (EETs) and Required Execution Times (RETs) can be used to specify (real-)time constraints (see Section 5.5.4)
M13	Support of operating systems	○	Not supported in this thesis (see M11).
M14	Integration of IP components	●	IP components can be integrated at the Virtual Target Architecture Layer and can be connected with Actor Sockets and Shared Object Sockets via signals (see Section 5.6.2).

CHAPTER 6

Simulation

6.1 Introduction

This Chapter describes the simulation model of the modeling and refinement methodology presented in Chapter 5. The simulation covers the presented modeling elements of the *Behavioral*, *Application* and *Virtual Target Architecture Layer* and the specified mapping and refinement steps. The implementation has been performed in C++ using the SystemC^{TM} event-driven simulation interface and library. The presented simulation library is called OSSS (Oldenburg System Synthesis Subset) simulation library and can be downloaded from `http://system-synthesis.org/download`.

Figure 6.1 gives an overview of the simulation library layers and mappings between them. The **Behavioral Layer** modeling elements are hierarchical behaviors and a pre-defined set of

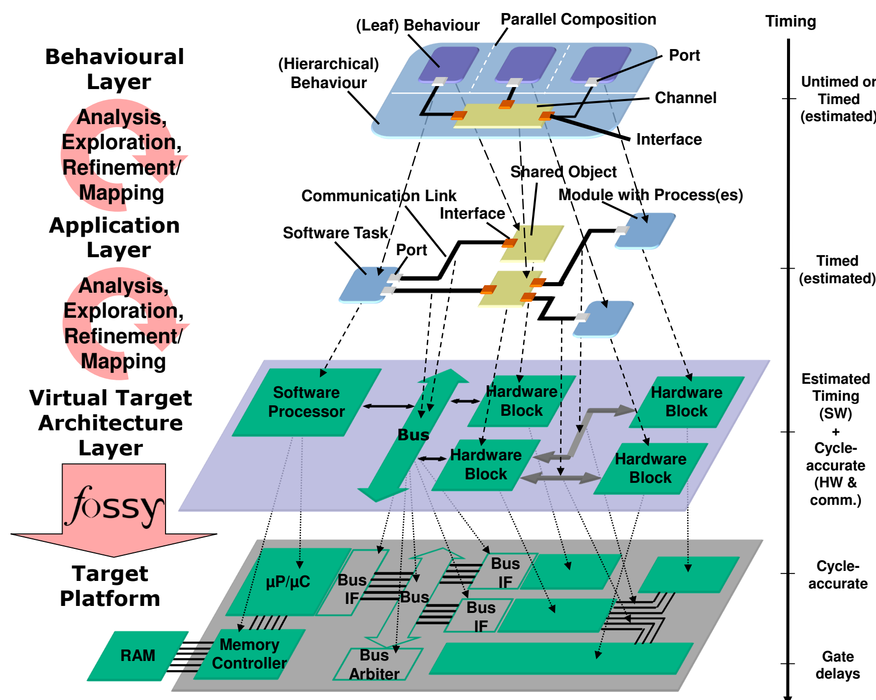


Figure 6.1: OSSS simulation library overview of modeling layers and mappings

communication channels that can be composed following the Program State Machine semantics as described in Section 5.4. The initial Behavioral Layer model is untimed and used to expose the maximum degree of parallelism the functional system description offers (or can be achieved with a reasonable amount of modeling effort). The OSSS Behavioral Layer simulation model elements are described in Section 6.3 in more detail. The mapping of an OSSS Behavioral model to an Application Layer model is described in Section 5.5.5.

The main modeling element at the **Application Layer** is called *Actor*. An Actor consist of a single or multiple processes (depending on its kind, either represented by a (sequential) *Software Task* or a parallel *Hardware Module*). Communication and synchronization between Actors is modeled by *Shared Objects*. These are special communication objects that provide a method interfaces for communication and guarantee a consistent access of an arbitrary number of concurrent Actors. The semantics of these Application Layer modeling elements is described in Section 5.5.

Functionality in Leaf Behaviors and Actors is described by a subset of C++. This subset is extended by hardware data types (i.e. bit vectors) under consistent involvement of object-oriented features that are supported by the simulation and synthesis infrastructure (see Section 6.2):

- Encapsulation of data and operations (methods) in classes. This is a basic object-oriented design principle for raising re-use.
- Method-based communication between structural blocks avoids effort of hand-crafted signal based communication and synchronization.
- Class inheritance allows easy extendability for re-use. Polymorphism, which is based on inheritance relations, can be used to express implementation alternatives.
- Template classes offer easy parametrization (e.g. buffer sizes) and make IP components more flexible.

In summary, at the Application Layer we specify the function, logical structure, and an approximate time response of the system. Profiling results from analysis of the Behavioral Layer model can be annotated to the Application Layer model to obtain an approximate-timed behavior. Also back-annotation approaches, where the execution of Leaf Behaviors are profiled on their expected target processing elements, are possible, but not further discussed here. Besides evaluation of the functionality of the system, the Application Layer offers an easy evaluation of design alternatives (e.g. HW/SW partitioning, scheduling, communication structures, and data locality). Profiling of different Application Layer model alternatives with regard to their performance can be accomplished easily since the component's allocations and scheduling can be changed quickly. Analysis of the executable Application Layer model in early design phases can help to detect and resolve bottlenecks in the logical structure. This might result in the relocation of timely critical computations from software to hardware or in reorganizing complex computations in pipeline structures to enhance the throughput.

The Application Layer Model is executable and abstracts from platform details regarding the communication between Actors and Shared Objects. The OSSS Application Layer simulation model elements are described in Section 6.4 in more detail.

The Application Layer gets further refined and mapped to a component model of the targeted implementation or execution platform. The **Virtual Target Architecture Layer** model adds implementation details of the target architecture with a special focus on the target platform inter processing element communication network. The semantics of the Virtual Target Architecture Layer modeling elements is described in Section 5.6. The OSSS Virtual Target Architecture Layer simulation model elements are described in Section 6.5 in more detail.

Communication links between Actors and Shared Objects of the Application Layer model are mapped onto cycle accurate communication channels. The mapping of the Application to the Virtual Target Architecture has already been described in Section 5.6.3. A flexible model for cycle accurate bus and point-to-point connections that enables the description of different protocols and data bandwidths enables exploring the impact of different protocols, data widths and arbitration schemes on the timing behavior of Application Layer model.

The resulting Virtual Target Architecture (VTA) model's communication is cycle accurate including the approximate timing annotation inside Actors. Moreover, the Application Layer

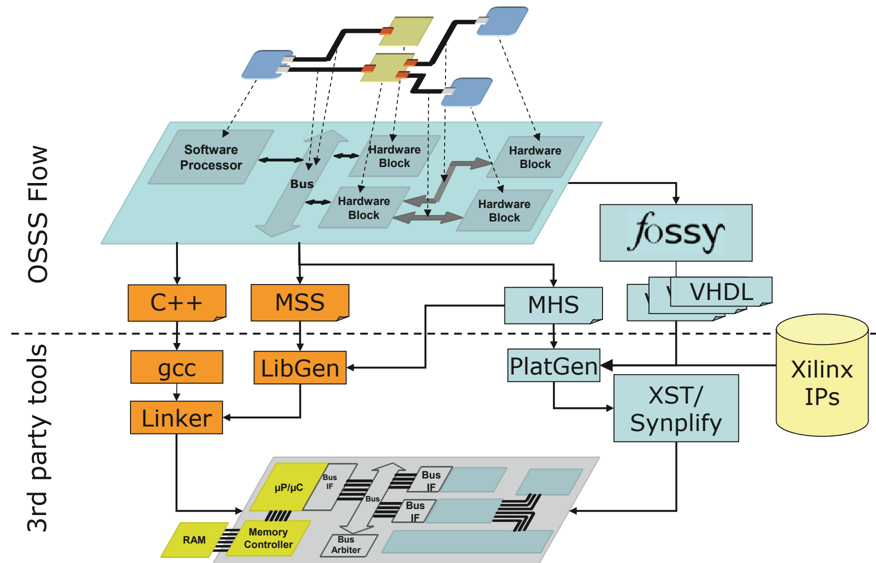


Figure 6.2: Overview of OSSS synthesis

mapped on the VTA Layer model is again an executable model that contains all relevant information required to start the back-end synthesis flow towards an implementation model.

In Section 6.6 the mapping of an executable Application Layer model to a Virtual Target Architecture Layer model, including a description of simulative architecture exploration, is described.

The VTA model is the input for the automatic synthesis process. It generates the overall system architecture, synthesizable VHDL for each hardware component, and C/C++ code for each software task. For the software parts a driver API and for the hardware parts a bus interface is automatically generated. Depending on the chosen platform, different so-called architecture description files can be generated. Special properties of different target platforms require adoptions of the synthesis process, e.g. for embedding special IP blocks or the generation of 3rd party tool specific configuration files. Figure 6.2 shows the overall synthesis process that has been tailored for Xilinx FPGA target technology.

The *Fossy* (**F**unctional **O**ldenburg **S**ystem **Sy**nthesiser) synthesis tool is capable of transforming the Shared Objects and Actors into hardware and software representations that can be further processed by vendor specific compilers and RTL hardware synthesis tools to end up with a physical implementation on the chosen *Target Platform*. More details about the synthesis can be found in Chapter 7.

6.2 Overview

6.2.1 SystemC™

Over the last years, SystemC™ [13] has gained much interest in academia and industry all over the world. The main reason for this popularity is its ease of application, extendability and flexibility. SystemC is not a language but a C++ class library implementing a discrete event simulator. Thus, the only additional "tools" a designer needs are his favorite text editor and a C++ compiler. In order to perform a simulation, the compiled design is executed on the workstation. No commercial HDL simulator is required. Furthermore, the class library approach makes it easy to use pre-existing C/C++ code in a SystemC model. This is very useful, especially for complex testbenches.

The main building blocks of SystemC are modules containing processes for modeling combinatorial and sequential circuits. Communication is performed through ports that are bound to either primitive or hierarchical channels. The built-in primitive signal channel enables modeling

of RT-level communication using delta-cycle update semantics. For more technical details on SystemC refer to the IEEE Std 1666-2011 Standard SystemC Language Reference Manual [13].

6.2.2 OSSS

OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) extends the synthesizable subset of SystemC [33] by adding constructs that facilitate the use of object-oriented features with well-defined simulation- and synthesis-semantics for the description of the hardware part of a Hardware/Software System on Chip. OSSS supports C++ concepts, like classes, templates, inheritance and the modeling elements of the Behavior Section 5.4, Application Section 5.5 and Virtual Target Architecture Section 5.6 Layers. OSSS is provided as an open-source C++ class library [225] which can be used in arbitrary SystemC models.

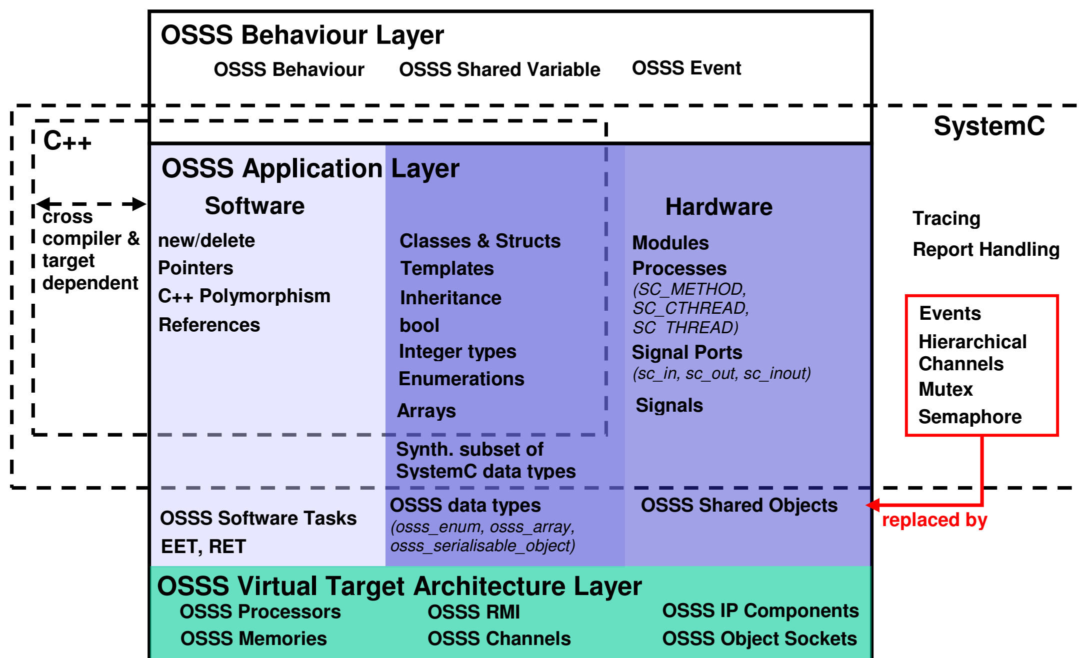


Figure 6.3: OSSS language subset overview

Figure 6.3 shows OSSS in comparison to C++ and SystemC. Technically, SystemC is a class library that builds on top of the C++ language. From a methodical point of view SystemC extends C++ by a notion of time, concurrent sequential processes, events and signals that are well known from hardware description languages like VHDL or Verilog. The communication concept with primitive and hierarchical channels has been directly adopted from SpecC. The latter point of view is the foundation of the diagram shown in Figure 6.3 where C++ is considered a subset of SystemC. In particular the OSSS covers

- the entire SystemC synthesizable subset (as defined in [33]),
- compiler and target independent C++ language constructs and some cross compiler and target processor dependent C++ language elements and standard libraries[1]

and extends SystemC by

- hardware/software implementation independent *Behaviors*, *Shared Variables* and *Event Channels* to cover the main system-level modeling elements of the *Behavioral Layer*,
- *Software Tasks*, *Estimated Execution Time (EET)* and *Required Execution Time (RET)* annotations, *Shared Objects*, and serializable data types to cover the modeling elements of the *Application Layer*,

[1]only tested for Xilinx MicroBlaze and PowerPC

- architectural building blocks like processors, memories, *Object Sockets*, *RMI Channels*, *OSSS Channels* and *IP Components* to cover the modeling elements of the *Virtual Target Architecture Layer*.

OSSS is subdivided into a hardware/software independent, a software only, a hardware only, a hardware/software intersection and a virtual target architecture part. In Figure 6.3 this is illustrated by different colors. In the following subsections these different layers with their most important language constructs will be briefly described.

6.2.3 Behavioral Layer

The *Behavioral Layer* provides modeling elements for capturing an embedded system on chip's functionality. The Behavioral Layer enables to compose sequential functions hierarchically through the following compositions: sequential (SEQ), finite state-machine (FSM), parallel (PAR), pipeline (PIPE).

Communication between these hierarchically composed functions (called *Behaviours*) is enabled through *Shared Variables* and *Events*. These can be combined and encapsulated in user-defined channels. The following pre-defined channels (using a combination of Shared Variables and Events) are provided: Queue, Handshake and Double Handshake.

A SystemC implementation independent description of the *Behavioral Layer* is presented in Section 5.4. More implementation details will be provided in Section 6.3.

6.2.4 Application Layer

The *Application Layer* provides modeling elements for HW/SW co-design of an embedded system on chip. The provided modeling elements allow restructuring of a Behavioral Layer model into hardware, software and communication specific elements. The modeling elements are *Software Tasks* for encapsulating functionality to be implemented in software, *Hardware Modules* for encapsulating functionality to be implemented in custom hardware and *Shared Objects* for communication and description of custom hardware accelerators.

To represent execution times of Software Tasks and Hardware Modules, a block annotation for sequential functional C/C++ code, called Estimated Execution Times (EETs), is provided. During model execution, timing constraints can be checked using Required Execution Timed (RET) block annotations.

A SystemC implementation independent description of the *Application Layer* is presented in Section 5.5. In the simulation model the introduced concept of *Actors* is implemented by *Software Tasks* and *Hardware Modules*. More implementation details will be provided in Section 6.4.

6.2.4.1 Hardware/Software Intersection

Modeling elements of the hardware/software intersection can be used in both, the hardware and the software domain. When considering hardware/software systems the data types of this intersection describe the backbone for hardware/software communication.

Composite data types like `struct`s and `class`es in both template and non-template versions (as known from C++) are allowed. Inheritance (i.e. specialization) which is a very important feature of object oriented design is also supported.

Data types are the basic C++ types like boolean (`bool`), integer (`signed char`, `unsigned char`, `char`, `signed/unsigned short`, `signed unsigned int`, ...), and floating point (`float`, `double`). No pointer, reference or "native" array types from C++ are supported in this intersection. The size of the supported basic types are strongly (cross) compiler dependent (for more details refer to Section 7.6). As compiler independent bit-true data types the synthesizable subset of SystemC data types[2] are supported as well. E.g. `sc_uint<9>` data type has a size of exactly 9 bits after synthesis.

Finally, special OSSS data types are provided:

- `osss_enum<nativeCppEnum>` serves as a wrapper for native C++ enumeration types (`enum`),

[2]For more information about the synthesizable data types of SystemC please refer to [33]

- osss_array<*dataType*, *Size*> serves as a wrapper for native C++ array types (e.g. unsigned int[16]) and
- osss_serialisable_object is a base class for all user-defined classes that need to be serialised into a bit-vector representation.

6.2.4.2 Hardware Section

The hardware section consists of two parts: The "traditional" synthesizable subset of SystemC and OSSS *Shared Objects*. The main structural hardware element is a module (sc_module) that may contain other modules (sub-modules), processes of kind SC_METHOD and SC_CTHREAD, signal ports of type sc_in<T>, sc_out<T> or sc_inout<T> (with T ∈ synthesizable data types) and ports to Shared Objects osss_port<osss_shared_if<IF> > (with IF ∈ inteface class of Shared Object). Ports can be bound to ports of sub-modules (when they have the same type), to signals sc_signal<T> of compatible type T and to Shared Objects implementing the required interface IF.

The modeling style of for custom hardware is behavioral RT. Either using sequential functional behavior with a run-to-completion semantics encapsulated in an SC_METHOD or step-/clock-wise execution of a sequential functional behavior encapsulated in an SC_CTHREAD (clocked thread). Wait statements are used to insert clock boundaries into the behavior description. For more details regarding the process modeling style of custom hardware, see Section F.2.2.

Some of the non-synthesizable SystemC features are *events*, h*ierarchical channels*, *mutexes* and *semaphores*. These language elements are very useful when designing on a higher level of abstraction, especially at a time where neither the target architecture nor the target platform has been specified. In OSSS, Events and user-defined channels for synchronization and communication are supported at the *Behavioral Layer* only. For synthesis a replacement for Events and user-defined channels (including Mutex and Semaphore) is provided through *Shared Objects*.

Shared Objects provide a mechanism to deal with method calls from concurrently executing processes and enable modeling of shared resources and abstract communication. They can be customized in different ways and provide a synthesizable replacement for SystemC Channels [22], Mutexes and Semaphores. Since they provide a simple guard mechanism that blocks a method call to a *Shared Object* until a certain condition evaluates to true, they can even be used to model events.

There are several other non-synthesisable features of SystemC like signal tracing and report handling. These features are either used in testbenches or are disabled for synthesis.

6.2.4.3 Software Section

In general the software section of Figure 6.3 contains the whole ISO C++ [14] language. Practically, the capabilities of different software (cross) compilers for the supported processing elements limit the software modeling subset.

In OSSS *Software Tasks* are the execution environment for the software part of a hardware/-software design. It can be regarded as the software counterpart of an sc_module. While an sc_module can contain multiple processes implemented in hardware, a Software Task contains exactly one single thread of control (i.e. representing the sequential software program running on a processor).

Timing properties of the software running on a processing element are modeled through EET source-level block annotations.

6.2.5 Virtual Target Architecture Layer

The *Virtual Target Architecture Layer* is organized as a an extendible *Virtual Target Architecture Class Library* containing building blocks that need to be instantiated and connected to assemble the execution platform consisting of processing elements, memories, custom hardware and communication networks (OSSS-Channels are used to connect processing elements, memories and user-defined hardware block with each other). The Application Layer model consisting of Software Tasks, Hardware Modules and Shared Objects is mapped onto the building blocks of the Virtual Target Architecture Layer. Communication of Software Tasks and Hardware

Modules with Shared Objects is implemented using a pre-defined Remote Method Invocation (RMI) protocol.

A SystemC implementation independent description of the *Virtual Target Architecture Layer* is presented in Section 5.6. More implementation details will be provided in Section 6.5.

6.3 Behavioural Layer[3]

The Behavioural Layer is the entry for the initial functional model. It provides all necessary means to capture a parallel hierarchical functional model of the application and supports different Models of Computation.

6.3.1 Introduction and motivation

One of the most prominent system design methodologies [160] is based on the SpecC language [181]. It introduces a minimal set of orthogonal language constructs that can be used for the description and refinement of embedded hardware/software systems.

The Behavioural Layer of OSSS implements the SpecC Program State Machine model (as described in Section 3.6.2) in SystemC. In [132] it has been shown that in principle the SpecC refinement methodology is applicable to SystemC designs as well. By adding SpecC modeling elements to SystemC major obstacles, as identified in [132], can be overcome:

(a) SystemC supports both static and dynamic scheduling. However a static scheduling allows a designer to determine the explicitly modeled execution sequence. This is very helpful during architecture exploration and refinement. SpecC supports only static scheduling using *par*, *pipe* and *fsm* constructs, or default sequential execution. These features are not available in SystemC. Moreover when static sensitivity is used for scheduling in SystemC it affects both the data transfer and the execution sequence (see (d)).

(b) SystemC uses *module* as the structural entity and *process* as the behavioural entity. It does only support hierarchical modeling of processes through explicit (dynamic) process spawning, which is rather tedious and error-prone. Especially the possibilities of unconstrained process spawning are a burden on the designer.

(c) In SystemC variables and events cannot be used to connect ports of different modules. Therefore, they can only be used inside modules or globally. This limits the use of events for scheduling modules and variables for data transfer between modules.

(d) In SystemC signals are used for data transfer between modules. The value of a signal is updated after a delta cycle. This concept makes it difficult to use signals in a specification model (cp. (a)). Therefore, designers should only use dynamic sensitivity for scheduling in specification models using SystemC. This again turns out to be a tedious and error-prone task.

Not only SystemC benefits from the the availability of SpecC modeling elements. Also SpecC, which is a super-set of C, benefits from SystemC/C++ in the following ways:

- User-defined data types and operators (encapsulation of data and operations on them),
- with access protection (public, protected, private),
- generic and template meta-programming,
- reuse and specialization through inheritance & polymorphism (can be applied to both data and behaviours),
- access to industry standard C++ libraries like STL and BOOST,
- and of course reuse of existing C/C++ code.

[3]This section is based on own previous work [43].

- For synthesis of the refined behaviour model state-of-the art SystemC synthesis tools can be used.

The implementation of PSMs based on SystemC should meet the following design principles:

(a) Compliant to the IEEE Std-1666™-2011 SystemC standard [13]. Thus, we restrict ourselves to build our extensions only upon the public classes mentioned in the standard. This also restricts us to the SystemC scheduler and process abstraction mechanisms. This imposes some limitations that will be discussed later.

(b) Our proposed extension should fit well into the general rules and coding styles for SystemC code. Especially the syntax of our extensions should fit seamlessly into the standardized SystemC API. While all SystemC language constructs carry the `sc` prefix we have chosen to use the `osss` prefix. Moreover, it should be possible to mix SystemC language constructs with our extensions wherever applicable.

(c) The semantics behind our extension should be equal to the PSM semantics implemented in SpecC. It is desirable to introduce a syntax that is as close as possible to SpecC. This should allow for easy porting designs from SpecC to SystemC using our proposed extensions. To some extent, this requirement is conflicting with the conformance to the SystemC syntax as required above. Considering this, we have tried to use the best from both worlds.

6.3.2 Composite Behaviours

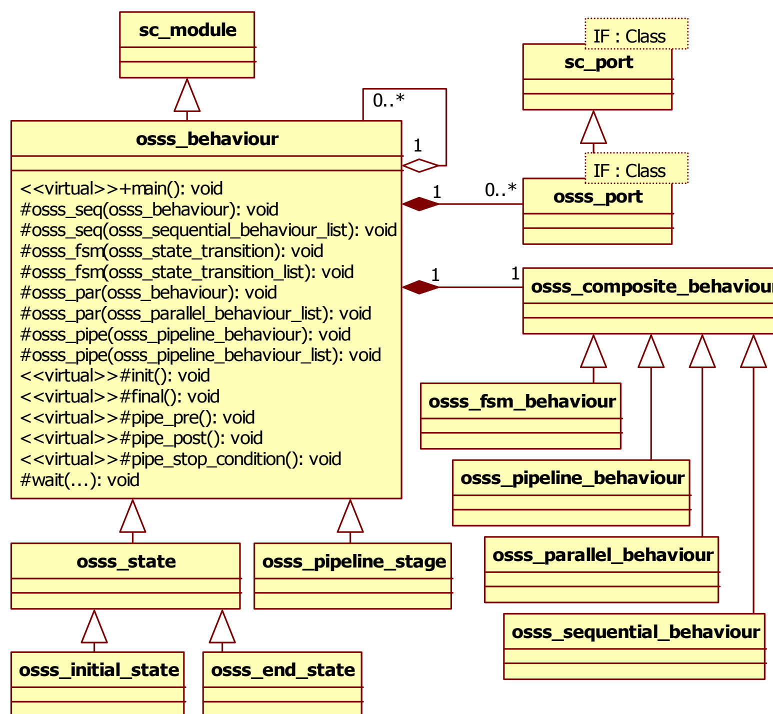


Figure 6.4: OSSS Behaviour class diagram (based on [43])

Figure 6.4 shows the OSSS Behaviour class diagram integrating composite behaviours into SystemC. The `osss_behaviour` class is derived directly from the main structural element of SystemC, the `sc_module` class. Other leaf behaviours, such as pipeline stages (behaviours that can be connected to implement a pipeline) and states (behaviours that can be used to build a finite-state machine) are directly derived from the main behaviour class.

Composite behaviours are implemented using the delegation pattern. The `osss_composite_behaviour` class is responsible for the handling of sequential, FSM, parallel, and pipelined composite behaviours.

The hierarchy relation itself is inherited from the `sc_module` class since each module is allowed to contain any number of child modules. Communication out of a behaviour is always performed through ports. To better distinguish behaviour-related ports from SystemC built-in ports we have simply derived our `osss_port` class from the SystemC `sc_port` implementation.

Pre-defined macros support the definition of behaviours (`OSSS_BEHAVIOUR`) and default constructors of behaviours (`BEHAVIOUR_CTOR`). This is in compliance to the `SC_MODULE` and `SC_CTOR` macros provided by SystemC. Corresponding macros are also provided for other leaf behaviours:

- `osss_state` is a regular state of a finite state machine composite behaviour (`OSSS_STATE`, `STATE_CTOR`).
- `osss_initial_state` is the initial/start state of a finite state machine composite behaviour (`OSSS_INITIAL_STATE`, `INITIAL_STATE_CTOR`). Only a single initial state per finite state machine is allowed.
- `osss_end_state` is the end/final state of a finite state machine composite behaviour (`OSSS_END_STATE`, `END_STATE_CTOR`). Multiple but a least one end state per finite state machine is allowed.
- `osss_pipeline_stage` is a pipeline stage of a pipeline composite behaviour (`OSSS_PIPELINE_STAGE`, `PIPELINE_STAGE_CTOR`)

The behaviour code itself has to be written into the body of the `main()` method. When a behaviour is entered or activated the `init()` method is executed once. When a behaviour is left or deactivated the `final()` method is executed once. The `init` and `final` hooks can be used to force certain code to be executed before and after the execution of the `main` routine.

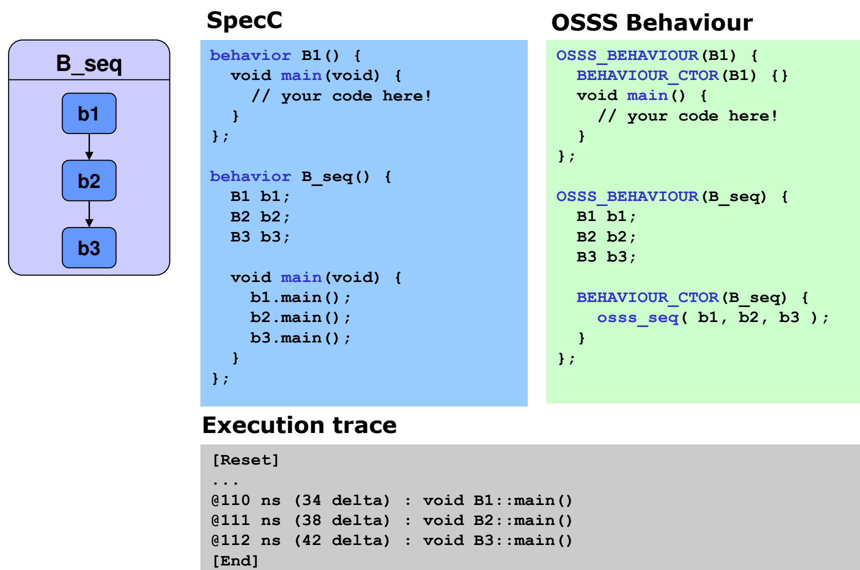


Figure 6.5: Sequential behaviour composition

A sequential behaviour composition (as shown in Figure 6.5) describes a purely linear control flow between sub-behaviours. Leaf behaviour `b1` is executed until its `main()` routine has been finished, then leaf behaviour `b2` is executed, and so on. `B_seq` is left when the last sequentially schedulued leaf behaviour (`b3`) has finished its execution.

This kind of composition corresponds to the linear sequential execution as known from any imperative programming language. In the finite-state machine behaviour in Figure 6.6 the execution order of the sub-behaviours depends on the evaluation of guards at the transition arcs. After the initial state `s1` has been entered the successor state `s2` can be entered when the transitions' guard expression `a<0` evaluates to true. `B_fsm` is left when `s4`, that implements an end state, has been entered.

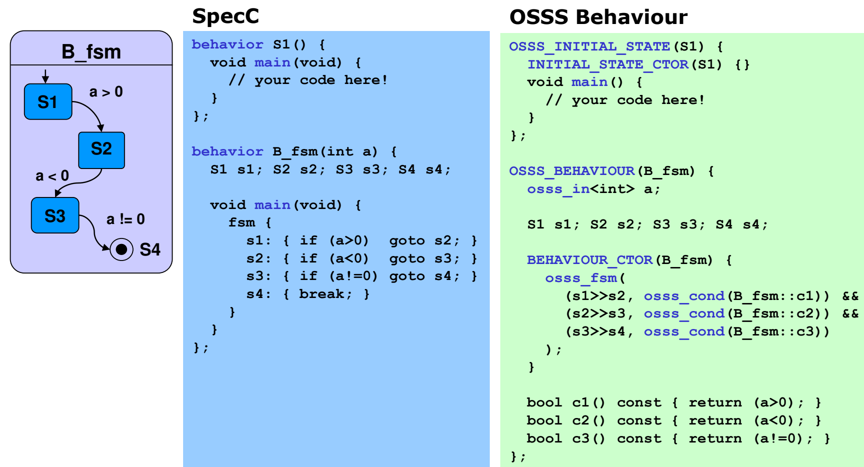


Figure 6.6: Finite State Machine behaviour composition

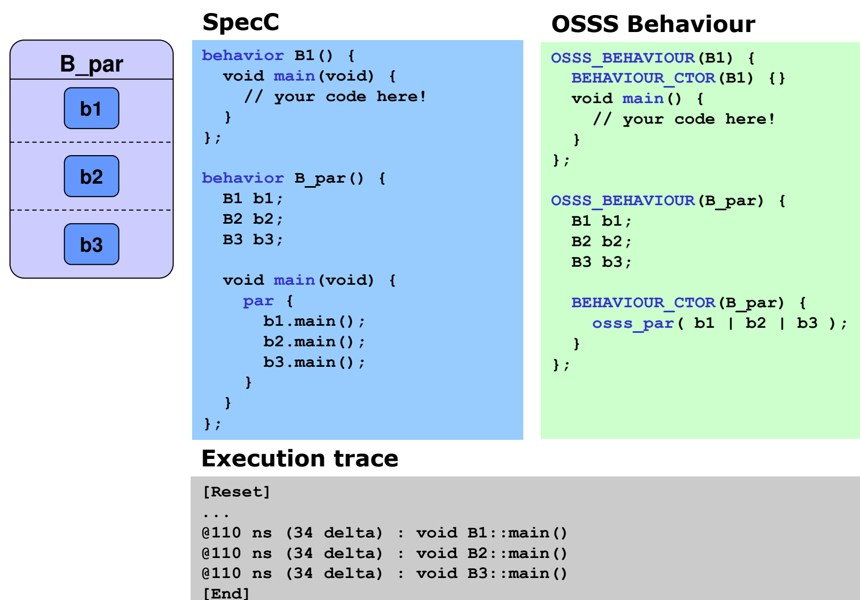


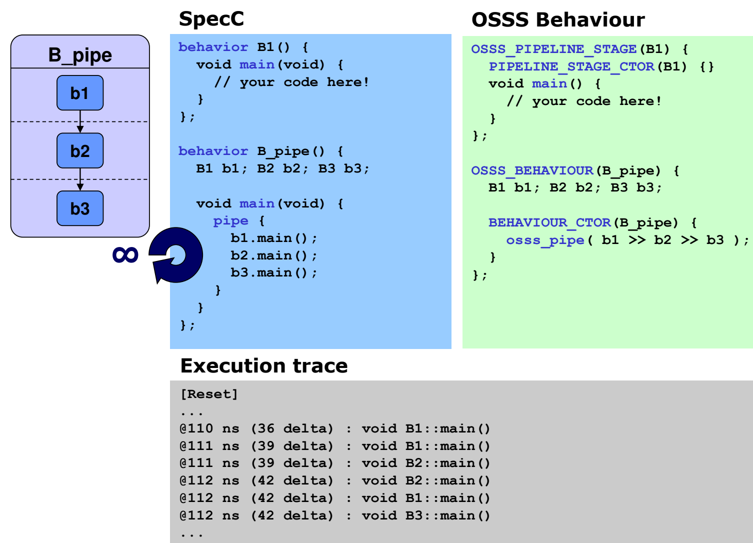
Figure 6.7: Parallel or concurrent behaviour composition

In a concurrent behaviour, all sub-behaviours become active whenever the parent behaviour is entered. In Figure 6.7 the sub-behaviours `b1`, `b2`, and `b3` are executed concurrently when `B_par` becomes activated. `B_par` is left when all concurrent sub-behaviours `b1`, `b2`, and `b3` have finished their executions. This corresponds to the general FORK-JOIN pattern known from parallel programming.

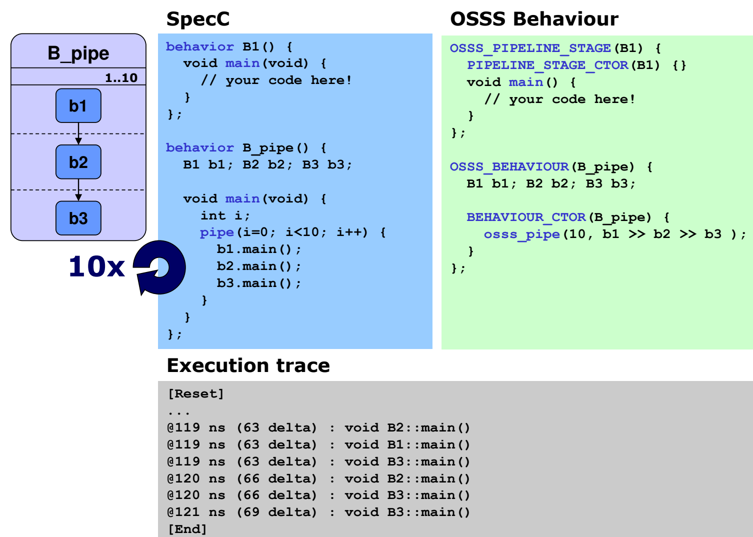
A combination of concurrent and sequential behaviours is the pipeline behaviour. In general a pipeline describes some sort of stream processing in which the same sequence of operations is performed on a stream of data. When `B_pipe` in Figure 6.8 is activated then `b1` starts its execution. When it is finished `b1` and `b2` execute in parallel until both of them are finished. In the next step `b1`, `b2` and `b3` execute in parallel until a certain stop condition evaluates to true. Otherwise a pipeline behaviour runs forever.

The `operator>>()` specifies the execution order of pipeline stages. Usually a pipeline executes in an endless loop as shown in Figure 6.8a. Execution of pipeline behaviours can be limited in two different ways:

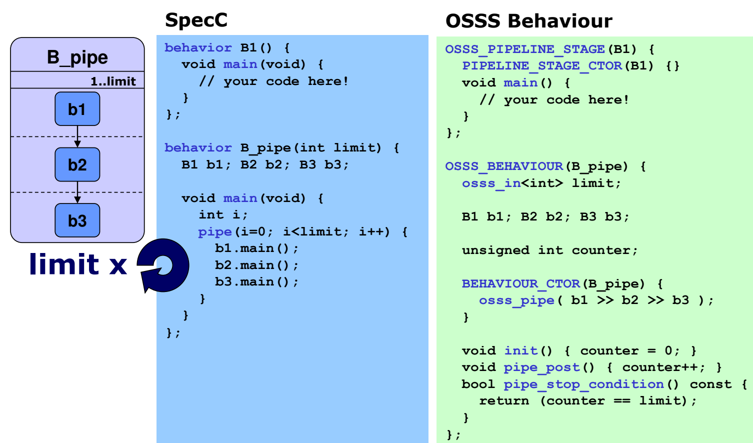
static For a fixed amount of executions for each pipeline stage the pipeline constructor `osss_pipe(...)` can be used with a constant that specified the amount of execution times.



(a) infinte execution



(b) statically bound execution



(c) dynamically bound execution

Figure 6.8: Pipelined behaviour composition

In Figure 6.8b the pipeline construction `osss_pipe(10, b1 >> b2 >> b3)` specified a pipeline whose pipeline stages are executed 10 times each.

dynamic If the number of pipeline executions can not be statically bound during design time, a stop condition that is evaluated during run-time, can be used. In Figure 6.8c the `init()` (executed before overall pipeline execution), `pipe_pre` (executed before each step of the pipeline), and `pipe_post` (executed after each step of the pipeline) in conjunction with the `pipe_stop_condition` hook can be used to dynamically limit the number of pipeline iterations. In Figure 6.8c the number of pipeline execution depends on the variable `limit` that is read from a shared variable outside `B_pipe`.

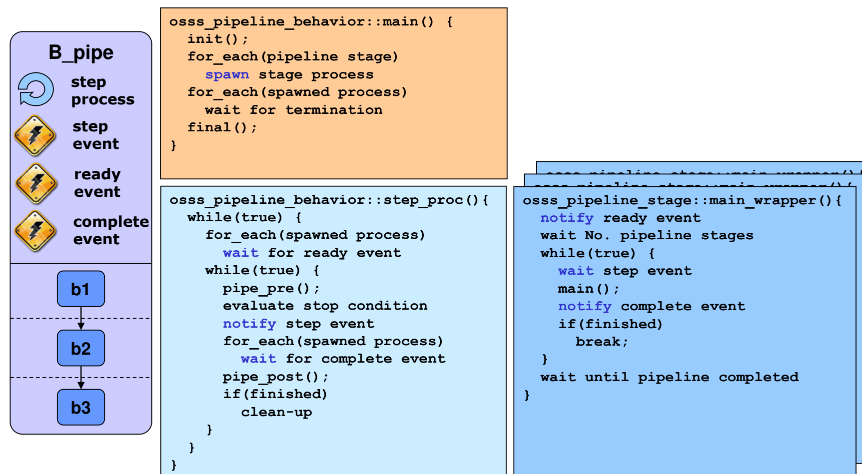


Figure 6.9: Implementation of pipelined behaviours in OSSS

Figure 6.9 depicts the implementation of pipelined behaviours in OSSS. Upon initialization of the pipeline, a process is created dynamically for each pipeline stage. The step process waits until all pipeline stages are ready for execution. If the stop condition of the pipeline does not evaluate to true a step event is notified. Each pipeline stage is sensitive to this step event and executes its functionality (encapsulated by the `main` function) for one time. During the ramp-up phase, each pipeline stage delays its execution of the `main` function until it is allowed to be executed. After execution of each pipeline stage the step process re-evaluates the stop condition and continues with the pipeline stage activation or terminates the pipeline.

6.3.3 Communication

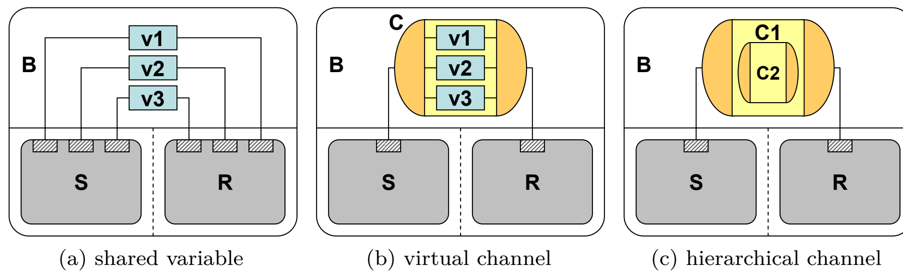
Until now, we have not talked about communication and data-dependent synchronization. Considering communication in the design of embedded systems along with a strict refinement process towards a physical implementation model, the following basic principles can be postulated:

(a) Separation of communication and computation.

(b) Declaration of abstract communication primitives.

(c) Enable custom communication implementation at different levels of abstraction.

SpecC provides channels for the explicit description of communication and thus enables the separation of communication and computation. A channel implements a certain interface that defines which communication primitives are provided. These abstract communication primitives can be used by behaviours that represent the computational parts of the design. Communication is initiated on ports that belong to a behaviour and can be accessed by its `main` routine. Only ports whose interface type matches with the interface implemented by a channel can be bound together and therefore establish a communication link. SystemC has adopted this design pattern directly from SpecC.



(a) shared variable (b) virtual channel (c) hierarchical channel

Figure 6.10: Communication in PSMs [150]

Figure 6.10 shows two different kinds of communication between behaviours. In Figure 6.10a shared variables `v1` to `v3` are bound to ports of behaviour `S` and `R`. Ports can be of direction *in*, *out*, and *inout*. This model of communication corresponds to shared memory communication.

In Figure 6.10b a virtual channel that encapsulates the communication using shared variables through a user-defined interface is shown. An example for such a virtual channel is a CSP-like "double handshake channel". This kind of communication corresponds to message passing communication. An extension of a virtual channel is the so-called hierarchical channel. This type of channel is allowed to contain virtual channels and is commonly used for the description of layered protocol stacks used in modern shared buses or network on chips.

A hierarchical channel as shown in Figure 6.10c is a virtual channel containing another channel. This kind of communication corresponds to a layered protocol stack as used with modern shared buses or network on chips.

Besides these data-oriented communication and synchronization primitives a notification concept based on events can be used as well. Events carry no data and occur only once in time after their notification.

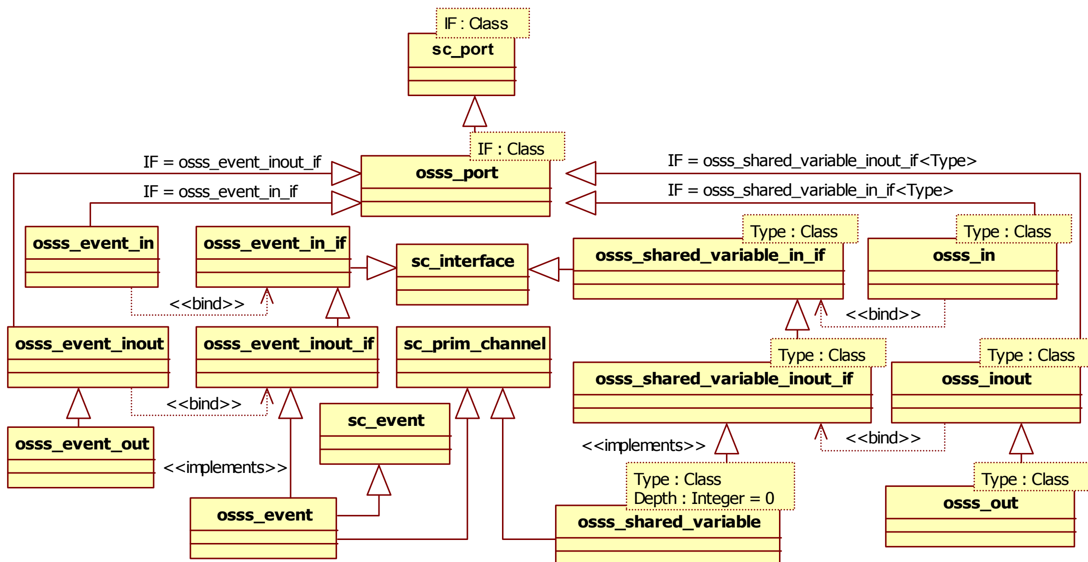


Figure 6.11: OSSS Behaviour communication class diagram

Before having a look at a specific example, we introduce the basic communication and synchronization primitives. Our implementation follows the "port-interface-channel" concept as shown in Figure 6.11.

Both channels `osss_event` and `osss_shared_variable<Type, Depth>` are derived from the SystemC primitive channel `sc_prim_channel` class. They implement different interface classes to enable read and write accesses. All of these interfaces are derived from the SystemC

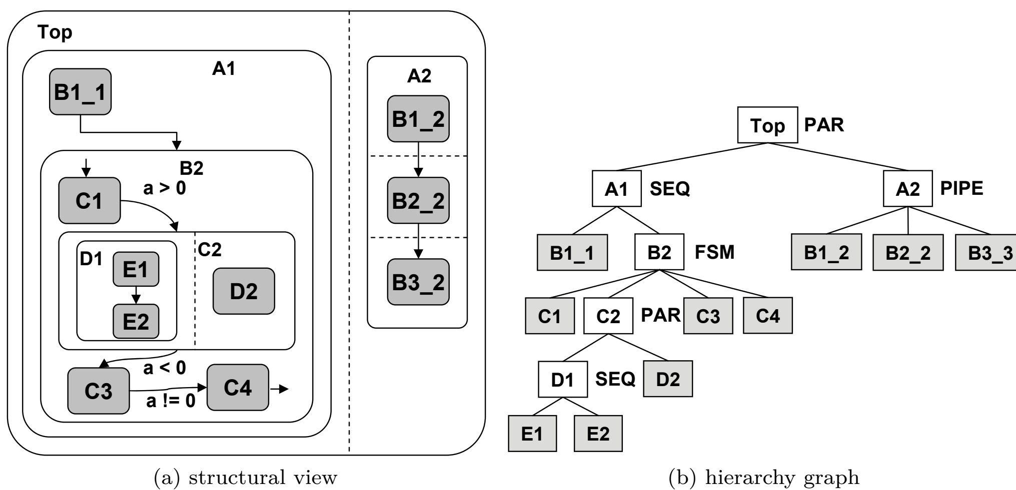
`sc_interface` class. Finally, port classes derived from `osss_port` that can be bound to different channel interfaces are supplied.

In SpecC shared variables can be of any type. Therefore, we have implemented them as a template container class. Shared variables have the advantage over SystemC `sc_signal`s that they do not follow the delta cycle update semantics.

When using shared variables for the communication between pipeline stages they need an internal delay representing the number of bypassed pipeline stages. Therefore we have introduced a second template parameter for specifying the depth of the *piped shared variable*. By default shared variables are not piped (i.e. depth = 0).

While shared variables are used in specification and early architecture models, signals are used inside channels in a communication model. Therefore, it is important to allow mixing of shared variables and SystemC signals in a model.

6.3.4 Hierarchical Behaviour composition



(a) structural view (b) hierarchy graph

Figure 6.12: Hierarchical PSM

Figure 6.12 gives an example of hierarchical behaviour composition. Figure 6.12a is a structural view depicting the hierarchy of beahviours. Figure 6.12b is the related hierarchy graph. Leaf behaviors (gray colored) implement the functionality inside their `main()` routines. Non-leaf behaviours are not allowed to implement any functionality.

```
1  OSSS_BEHAVIOUR(D1) {
2   protected:
3    E1 e1; // sub−behaviour e1
4    E2 e2; // sub−behaviour e2
5
6   public:
7    BEHAVIOUR_CTOR(D1) { osss_seq(e1, e2); } // sequential composition
8  };
9
10 OSSS_STATE(C2) {
11  protected:
12   D1 d1;
13   D2 d2;
14
15  public:
16   STATE_CTOR(C2) { osss_par(d1 | d2); } // parallel composition
17 };
18
19 OSSS_BEHAVIOUR(B2) {
20   osss_in<int> a;
21
22  protected:
23   C1 c1;
```

```
24    C2 c2;
25    C3 c3;
26    C4 c4;
27
28   public:
29    BEHAVIOUR_CTOR(B2) {
30      // Finite−State Machine (FSM) composition
31      osss_fsm(
32        // change from state c1 to c2,  iff  cond1 is true
33        (c1 >> c2, osss_cond(B2::cond1)) &&
34        // change from state c2 to c3,  iff  cond2 is true
35        (c2 >> c3, osss_cond(B2::cond2)) &&
36        (c3 >> c4, osss_cond(B2::cond3)) &&
37        (c3 >> c1, osss_cond(B2::cond4))
38    );
39    }
40
41    // conditions for use with state  transitions  above
42    bool cond1() const { return (a > 0); }
43    bool cond2() const { return (a < 0); }
44    bool cond3() const { return (a != 0); }
45    bool cond4() const { return (a == 0); }
46  };
47
48  OSSS_BEHAVIOUR(A1) {
49    osss_in<int> in_port;
50
51   protected:
52    B1_1 b1_1;
53    B2 b2;
54
55   public:
56    BEHAVIOUR_CTOR(A1) {
57      b2.a(in_port); // port to port binding: port a of behaviour b2
58                     // is bound to the in_port of this behaviour.
59      osss_seq(b1_1, b2);
60    }
61  };
62
63  OSSS_BEHAVIOUR(A2) {
64    osss_out<int> a_out;
65
66   protected:
67    B1_2 b1_2;
68    B2_2 b2_2;
69    B3_2 b3_2;
70    unsigned int counter;
71
72    osss_shared_variable<int, 1> v0; // piped (shared) variable
73    osss_shared_variable<int, 1> v1;
74
75    // The following code is used to constrain the number of pipeline
76    // executions for b1_2, b2_2, and b3_2. By default pipelines are
77    // executed as an infinite loop. In this case the pipeline is
78    // executed COUNTER_LIMIT times.
79    virtual void init() { counter = 0; }        // Before the pipeline  starts
80                                                // the counter variable is  initialised ,
81    virtual void pipe_post() { ++counter; }  // it is incremented after each
82                                                // pipeline execution,
83    virtual bool pipe_stop_condition() const // and the stop condition has been
84      { return (counter == COUNTER_LIMIT); } // reached on COUNTER_LIMIT.
85
86   public:
87    BEHAVIOUR_CTOR(A2) {
88      b1_2.a_out(v0);
89      b2_2.a_in(v0);
90      b2_2.a_out(v1);
91      b3_2.a_in(v1);
92      b3_2.a_out(a_out);
93      osss_pipe(b1_2 >> b2_2 >> b3_2); // pipeline composition
94    }
```

```
95  };
96
97  OSSS_BEHAVIOUR(Top) {
98   protected:
99    osss_shared_variable<int> var; // a regular (non-piped) shared variable
100
101   A1 a1;
102   A2 a2;
103
104  public:
105   BEHAVIOUR_CTOR(Top) {
106     a1.in_port(var);
107     osss_par(a1 | a2);
108   }
109 };
```

Listing 6.1: Composite behaviours of Figure 6.12

Listing 6.1 contains all composite behaviours from the example in Figure 6.12. The `Top` behaviour (line 97) contains the two sub-behaviours `a1` of type `A1` and `a2` of type `A2`. The `osss_par` process constructor call performs the parallel composition of behaviour instances `a1` and `a2`. A shared variable `var` of type `int` is used for communication between `a1` and the environment (not shown in the listing). It is bound to the port `in_port` of `a1` (line 99).

Behaviour `A2` (line 63) specifies a pipelined computation using the `osss_pipe` process constructor (line 93). The `operator>>()` specifies the execution order of pipeline stages. Usually a pipeline executes in an endless loop. The `pipe_pre` and `pipe_post` (executed before and after each step of the pipeline) in conjunction with the `init` and `pipe_stop_condition` hooks can be used to manipulate the number of pipeline iterations. In `A2` we have specified a simple for-loop that executes the pipeline exactly `COUNTER_LIMIT` times.

For the communication between pipeline stages we use pipelined shared variables with a depth of one, since each of them only crosses a singe pipeline stage. The piped shared variables are declared in lines 72 and 73 and bound to the ports of the three pipeline stage in lines 88-91.

Behaviour `A1` (line 48) describes the sequential composition of behaviours `B1_1` and `B2` using the `osss_seq` process constructor.

In behaviour `B2` (line 19) a finite-state machine is specified using the `osss_fsm` process constructor. It takes state transition definitions that are concatenated to a list using the `operator&&()`. The state transition list `(c1 >> c2, osss_cond(B2::cond1))` has the following semantics: A transition from behaviour `c1` to `c2` can be taken when the condition `B2::cond1` evaluates to true. The condition is implemented by a member function of type `bool(void) const`.

6.3.5 Current Limitations

During the implementation of PSMs we have encountered some limitations of the chosen approach:

(a) Our approach does not support any of the exception mechanisms (trap and interrupt) available in SpecC. This restriction is due to the underlying non-preemptive SystemC scheduler. Ways to overcome this limitation are described in [59, 159]. However, it requires a major change to the SystemC kernel that might become an extension to the current standard in the future[4].

(b) The former restriction imposes that we do only support the transition-on-completion (TOC) but no transition-immediate (TI) arcs in the finite-state machine implementation.

(c) SystemC is a C++ class library but no language. It imposes the usage of preprocessor directives (called macros) and restricts us to the syntax of C++ in general. This becomes rather cumbersome when dealing with complicated compiler error messages that rather bare the complexity of compilers than helping the designer. This can be mitigated by implementing various run-time design rule checks which present more useful information.

[4] In SystemC 2.3.0 an exception mechanism based on [59] has been implemented. At the time of the implementation of the OSSS Behavior simulation library this SystemC 2.3.0 has not been publicly available.

(d) Debugging can become cumbersome since PSMs may involve parallel behaviour that is implemented using a low-level thread/process model (such as POSIX). But this is a general problem existing for all multi-threaded applications including SpecC. A language or thread/process model aware debugger can be used to overcome this restriction.

6.4 Application Layer[5]

The Application Layer in OSSS enables the hardware/software partitioning and modeling of mixed hardware/software systems without explicit resource binding to processing elements and communication resources. On this layer the simulation library provides elements to model hardware modules and software tasks using (passive) objects and *Shared Objects* which can be shared between hardware modules and software tasks in the design (see Figure 6.13).

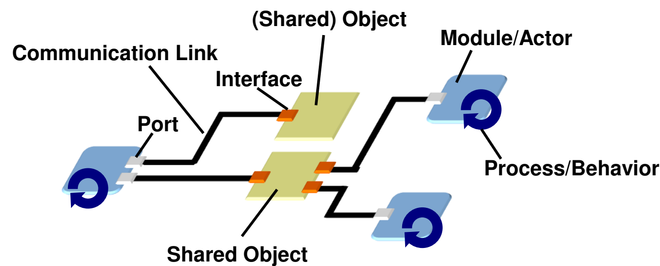


Figure 6.13: The *Application Layer* and its components (based on [44])

Hardware modules and software tasks are active components, i.e. they contain processes and thus have an own thread of execution. *Shared Objects* are passive which means they do not initiate any execution on their own. Shared Objects (see Section 6.4.1) have two major properties: on one hand they provide a method based interface for inter-process communication and on the other hand they are a scheduled shared resource which can be used by different processes.

Figure 6.14 illustrates a simple producer/consumer design at the Application Layer. The FIFO (First-In-First-Out) buffer between the producer and the consumer processes is implemented using a *Shared Object*. This simple design will be used to illustrate the use of Shared Objects at the OSSS Application Layer throughout this chapter.

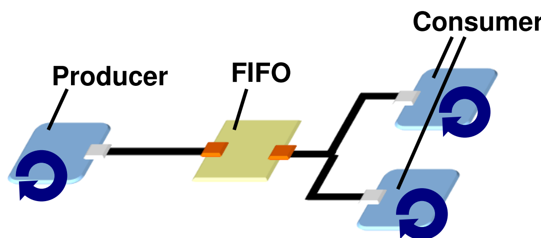


Figure 6.14: Producer/Consumer design on *Application Layer* [44]

6.4.1 Shared Object

A new concept introduced by OSSS is the Shared Object `osss_shared<C, S>`, with the user-defined class `C` and the pre-defined or user-defined scheduler `S`. While Shared Objects have no counterpart in C++, they can be used as a synthesisable replacement to model SystemC channels [22], mutexes, semaphores, and events (see red box in Figure 6.3). They are called "shared" because their state and behaviour is shared between different concurrent processes. Shared

[5]This section is based on own previous work [44].

Objects are similar to the monitor concept [199] used in some concurrent software description languages, because accesses are always mutually exclusive and arbitrated by a scheduler.

At the *Application Layer* an abstraction from the execution platform processing elements and the communication details is given. Communication between modules or tasks and Shared Objects is performed by user-defined method calls using message passing through communication links. Argument and return types are of any supported C++ data type (no pointers and references allowed). The communication link has the following properties:

directed: The source of a communication link is called a port while the destination is called an interface. Please note, that this does not necessarily mean that the data flow is from the port to the interface only. Since methods with both, a void and a non-void return value can be called, the data flow is bidirectional.

blocking: A method call to a port does not return until the called method has been completed.

A Shared Object implements at least one interface and only reacts to external method calls on its implemented interfaces. These methods are called *guarded* because access to them is optionally[6] restricted by a so called *guard expression*. An access to such a method is only granted if this expression evaluates to true and the arbiter returns permission.

The behaviour of a Shared Object as well as the scheduling algorithm of the Shared Object's arbiter is custom-designed. Due to this flexibility Shared Objects can be used for a variety of different purposes:

- Interprocess communication and synchronisation.
- Method interface for hardware modules.
- Modelling shared resources.

We call a process which contains a request to a method of a particular Shared Object a *client* or *client process* of that object. Likewise, we may also refer to a Shared Object as a *server*.

6.4.1.1 Using Shared Objects

To gain a better understanding of how Shared Objects are used in Application Layer models we are going to take a closer look at a simple consumer/producer example. The idea of this design example is to use a Shared Object as a container for a user-defined FIFO class. The FIFO class provides a method interface for accessing a memory in a First-In-First-Out style. The Shared Object around the FIFO class enables resource sharing by providing arbitration facilities for multiple concurrent accesses. The structure of this example is shown in Figure 6.14.

Instantiation and binding of Shared Objects

Figure 6.15 shows a more detailed structure of the producer/consumer example at the *Application Layer*. A producer implemented as a software task calls methods defined in the `FIFO_put_if` on a local port, which is bound to a buffer Shared Object. Two hardware consumer processes call methods defined in the `FIFO_get_if`. The user-defined `FIFO` class inside the Shared Object container implements the put and get interface. The behaviour of the Shared Object instance is determined by the user-defined `FIFO` class. It is specified to store 10 items of type `Packet`. A scheduling class (e.g. the pre-defined `osss_round_robin` or any other user-defined scheduling class) arbitrates concurrent accesses to the Shared Object containg the `FIFO`.

Listing 6.2 shows the OSSS Application Layer top-level design of the producer/consumer example as depicted in Figure 6.15. The communication links between the components are established by port to interface bindings: the output port of the producer (line 26) and both input ports of the consumer processes (line 33) are bound directly to the buffer Shared Object.

OSSS Shared Objects have two predefined ports. The `clock_port` and the `reset_port`. Both ports need to be bound to the global clock an reset signal of the design.

[6] a guard can be set to `true`

Notice: The usage of clock and reset ports at the Application Layer is mandatory due to the usage of the `SC_CTHREAD` processes. This is a pure technical restriction of the current OSSS implementation. A replacement with `SC_THREAD` is desirable, but currently not available.

```
#define OSSS_BLUE // Application Layer Model
#include <osss.h>
#include "Packet.hh"
#include "FIFO.hh"
#include "Producer.hh"
#include "Consumer.hh"

SC_MODULE(Top) {
  sc_in<bool> clk, reset;

  typedef osss_shared<FIFO<Packet, 10>,
                      osss_round_robin> Buffer_t;

  Producer   *m_Producer;
  Buffer_t   *m_Buffer;
  Consumer   *m_Consumer[2];

  SC_CTOR(Top) {
    m_Buffer = new Buffer_t("m_Buffer");
    m_Buffer->clock_port(clk);
    m_Buffer->reset_port(reset);

    m_Producer = new Producer("m_Producer");
    m_Producer->clock_port(clk);
    m_Producer->reset_port(reset);
    m_Producer->output(*m_Buffer);

    m_Consumer[0] = new Consumer("m_Consumer0");
    m_Consumer[1] = new Consumer("m_Consumer1");
    for(unsigned int i=0; i<2; ++i) {
      m_Consumer[i]->clk(clk);
      m_Consumer[i]->reset(reset);
      m_Consumer[i]->input(*m_Buffer);
    }
  }
};
```

Listing 6.2: Top-Level module of the producer/consumer example on Application Layer

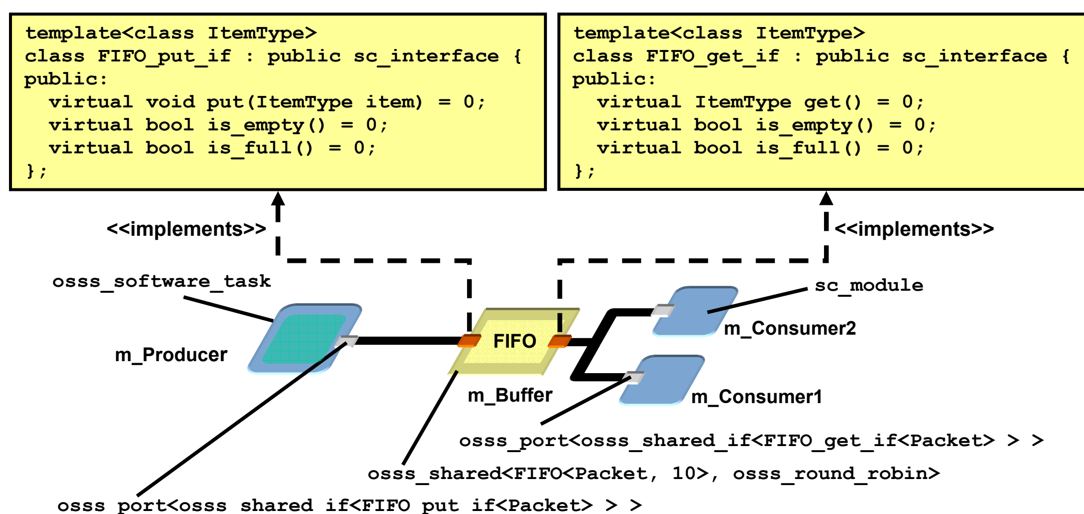


Figure 6.15: Producer/consumer example at the *Application Layer* [44]

Declaration of Shared Objects

In the following it is shown how a user-defined class, like the `FIFO`, has to be implemented to be usable as a Shared Object. First of all an abstract interface class needs to be specified. This abstract interface class specifies services the Shared Object provides for its attached client processes. It is possible to have more than a single interface per Shared Object.

Listing 6.3 shows two abstract interface classes (`FIFO_put_if` and `FIFO_get_if`), and the FIFO class implementation itself. The interfaces need to be defined separately from their implementation and need to be derived from the SystemC interface class `sc_interface`. Interfaces of Shared Objects are pure virtual, i.e. they consist out of pure virtual methods and do not contain any data members.

The FIFO class is derived from both interfaces, the `FIFO_put_if<...>` (line 20) and the `FIFO_get_if<...>` (line 21). Since the FIFO class is not allowed to contain any virtual methods it needs to implement all method derived from these interfaces. In a user class of a Shared Object all exported (i.e. public accessible) methods are called *guarded methods*.

All guarded methods are implemented using special `OSSS_GUARDED_METHOD` macros (e.g. line 30) to specify the method signature together with its associated guard condition (last argument). If no guard is specified the guard condition is constantly set to true (e.g. line 45). A guarded method used inside a guard condition needs to be wrapped by the `OSSS_EXPORTED` macro (e.g. line 31).

The internal storage of the FIFO is described using an `osss_array` (line 60), which is a bounded vector of a user-defined type that can be converted easily to a physical memory (e.g. a Xilinx Block-RAM) during architecture refinement.

```
1  typedef unsigned int FIFO_size_t;
2
3  template<class ItemType>
4  class FIFO_put_if : public virtual sc_interface {
5  public:
6    virtual void put(ItemType item) = 0;
7    virtual bool is_empty() = 0;
8    virtual bool is_full() = 0;
9  };
10
11 template<class ItemType>
12 class FIFO_get_if : public virtual sc_interface {
13 public:
14   virtual ItemType get() = 0;
15   virtual bool is_empty() = 0;
16   virtual bool is_full() = 0;
17 };
18
19 template<class ItemType, FIFO_size_t Size>
20 class FIFO : public FIFO_put_if<ItemType>,
21              public FIFO_get_if<ItemType>
22 {
23  public:
24
25   FIFO() : m_buffer(),
26            m_put_index(0),
27            m_get_index(0),
28            m_num_items(0) {}
29
30   OSSS_GUARDED_METHOD_VOID(put, OSSS_PARAMS(1, ItemType, item),
31                            !OSSS_EXPORTED(isFull()) ) {
32     m_buffer[m_put_index] = item;
33     increment_index(m_put_index);
34     m_num_items += 1;
35   }
36
37   OSSS_GUARDED_METHOD(ItemType, get, OSSS_PARAMS(0),
38                       !OSSS_EXPORTED(isEmpty()) ) {
39     ItemType result = m_buffer[m_get_index];
40     increment_index(m_get_index);
41     m_num_items -= 1;
42     return result;
```

```
43     }
44
45     OSSS_GUARDED_METHOD(bool, is_empty, OSSS_PARAMS(0), true) {
46       return m_num_items == 0;
47     }
48
49     OSSS_GUARDED_METHOD(bool, is_full, OSSS_PARAMS(0), true) {
50       return m_num_items == Size;
51     }
52
53   protected:
54
55     void increment_index(FIFO_size_t &index) {
56       if (index == (Size-1)) index = 0;
57       else index += 1;
58     }
59
60     osss_array<ItemType, Size> m_buffer;
61
62     FIFO_size_t m_put_index, m_get_index, m_num_items;
63 };
```

Listing 6.3: FIFO interface and FIFO class implementation

Comparing the FIFO class from Listing 6.3 with a common C++ class implementation the main difference in the use of `OSSS_GUARDED_METHOD_VOID`, `OSSS_GUARDED_METHOD` and `OSSS_EXPORTED` constructs. These macros are provided by the OSSS library to bind a guard condition to a method.

In principle, any kind of user-defined C++ class can be used as a Shared Object. With the only restriction: Each method which should be accessible by client processes needs to be declared in an abstract interface class and implemented as a guarded method.

A C++ method declaration with a `void` return type of the form:

```
void methodName(paramType1 param1, ... paramTypeN paramN)
```

translates into:

```
OSSS_GUARDED_METHOD_VOID( methodName,
                          OSSS_PARAMS(  N,
                                        paramType1, param1,
                                        ...
                                        paramTypeN, paramN),
                          guardCondition)
```

A C++ method with a non-`void` return type of the form:

```
return_type methodName(paramType1 param1, ... paramTypeN paramN)
```

translates into:

```
OSSS_GUARDED_METHOD( return_type,
                     methodName, OSSS_PARAMS( N,
                                              paramType1, param1,
                                              ...
                                              paramTypeN, paramN),
                     guardCondition)
```

The main benefit of a Shared Object is that several clients can access the methods of that object without knowing about concurrent accesses from other clients. Thus, it is easy to add and remove clients of Shared Object without changing the Shared Object itself. Furthermore, the guard conditions can be used to implement an implicit protocol; that is, to control the order of accesses for the inquiring clients.

Communication with Shared Objects

Communication with Shared Objects follows the SystemC IMC (Interface Method Call) mechanism. It consists of

Port-Interface Binding: For the establishment of a *Communication Link* a port of a module or software task needs to be bound to a Shared Object. This binding requires a port of the same type as the interface provided by the object. For calling methods on a Shared Object which implements the interface class `IF`, a port of type `osss_port<osss_shared_if<IF> >` needs to be bound.

Method Call: When the port is bound to a Shared Object is acts like a constant reference. Using the `operator->()` on the port allows calling each method which has been declared by the interface class. As mentioned before, method calls to Shared Objects are blocking. They do not return control to the caller until the called method has been executed completely.

The schedulers

Concurrent accesses to guarded methods of a Shared Object are handled by a scheduler. The scheduling algorithm of a Shared Object can be changed easily by replacing the scheduler class. The OSSS-Library contains some pre-designed schedulers, these are listed in Table 6.1. Additional user-defined scheduling algorithms can be implemented easily.

Scheduler	Description	Algo
`osss_round_robin`	• No priorities • Fairness not guaranteed	3
`osss_modified_round_robin`	• No priorities • Fairness not guaranteed	4
`osss_static_priority<ZeroIsHighestPrio>`	• Static priorities • Default: Zero is lowest priority • Fairness not guaranteed	1
`osss_ceiling_priority<MaxClients>`	• Dynamic priorities • Fair	2
`osss_least_recently_used<MaxClients>`	• Dynamic priorities • Fair	2

Table 6.1: Schedulers included in OSSS

For scheduling algorithms that support priorities these can be set by passing a positive number to the `setPriority()` method of an `osss_port<osss_shared_if<IF> >` during SystemC elaboration phase. The interpretation of this positive number (e.g. higher number means higher priority) is scheduling algorithm dependent.

Custom scheduling algorithms are implemented by deriving from class `osss_scheduler`. The derived class needs to implement the `PositiveNumber schedule( const RequestVector & clientRequests )` method and requires to be purely combinatorial, meaning it must not contain any `wait()` statement.

Restrictions when using Shared Objects

Usage of Shared Objects is subject to the following restrictions:

SO-R01: Shared Objects must implement a default constructor.

SO-R02: All methods accessible from outside the Shared Object must be guarded.

SO-R03: Direct access to data members is not possible.

SO-R04: Shared Objects are passive and only react to requests from clients.

SO-R05: Calls to guarded methods are blocking.

SO-R06: Guarded methods are not allowed to be `const`.

SO-R07: Parameters of guarded methods are not allowed to be of a pointer (`*`) or a reference (`&`) type.

SO-R08: Parameters of guarded methods are not allowed to be `const`.

SO-R09: Guard expressions must be free of side effects, they must not change the inner state of the Shared Object.

SO-R10: Guards are only allowed to be dependent on the internal state of the Shared Object. I.e. the parameters of a guarded method are not allowed to be used in the associated guard evaluation.

SO-R11: The evaluation of a guard expression must not cause the execution of a `wait()` statement.

SO-R12: The guard expression must be satisfiable, meaning it must not be hardwired to false.

As for the Shared Object some restrictions apply to the clients that use Shared Objects:

SO-CL-R1: All client processes of a Shared Object must be driven by the same clock.

SO-CL-R2: Calls to methods of a Shared Object must be done by the `operator->()` method of the `osss_port<osss_shared_if<IF> >`. To complete this call a `wait()` statement has to follow after the `operator->()` call.

SO-CL-R3: Each `osss_port<osss_shared_if<IF> >` of a `sc_module` is allowed to be used by a single process only. If an `osss_port<...>` bound to a Shared Object is used by more than one process the simulation produces an error and is aborted immediately.

SO-CL-R4: Processes from which calls to Shared Objects originate must be `SC_CTHREAD`s.

SO-CL-R5: Parameters must be passed as values not references for the call to be synthesizable.

6.4.2 Adapter Socket

Adapter Sockets (short: Sockets) allow access to a low level signal interface from within Shared Objects. The purpose of sockets depends on whether they are used in combination with Shared Objects. If the socket interface is used together with a Shared Object it is possible to design a module that uses a method-based communication interface on one side and a signal interface on the other side. This kind of method interface to signal level interface adapter is also called *transactor*.

In the methodology (see Chapter 5) the possibility to access signals from within a Shared Object has been introduced at the Virtual Target Architecture Layer (see Section 5.6.2). In the OSSS simulation library, Adapter Sockets belong to the Application Layer. This allows to prepare the integration of signal level components and RTL IP components already at Application Layer.

6.4.2.1 Using sockets

The usage of sockets with a signal level interface will be described by the example of a shared memory adapter. As shown in Figure 6.16, it consists of a memory module, an adapter module - which is implemented as a Shared Object - and multiple clients.

The "RAM" module is implemented as a normal SystemC module and does not contain any constructs specific to sockets, so it will not be described in detail. It implements a signal interface that contains data-, address- and control signals.

The implementation of the clients also does not contain any constructs specific to sockets. They implement an interface to a Shared Object. In the next section the "SharedMemAdapter" module is described in-depth.

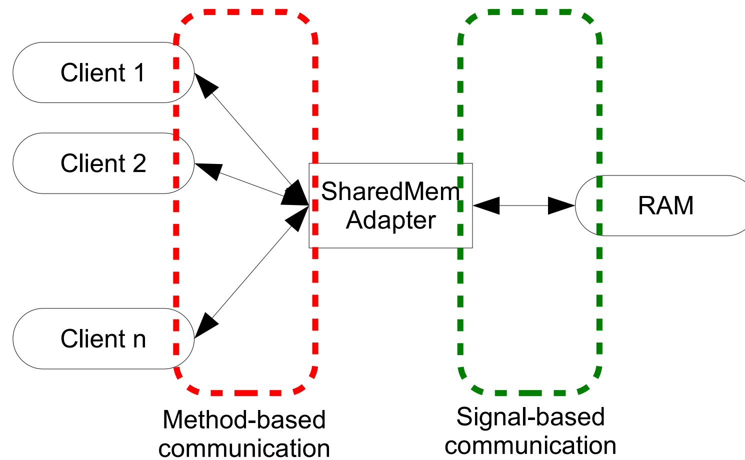


Figure 6.16: Signal adapter socket example [44]

Declaration of a socket

Basically the "SharedMemAdapter" module is declared like a Shared Object. The following listing shows the declaration of the module.

The only difference to the declaration of a Shared Object is the `OSSS_REQUIRED(if_name){...}` section. In this section the signals that make up the interface are declared the way a module port declaration is done in SystemC.

```
#include <osss.h>

const unsigned int AddrWidth=4u;
const unsigned int DataWidth=8u;

// This class will be used as shared object and acts as an interface
// to a memory
class SharedMemAdapter
{
  public:

    typedef sc_uint<AddrWidth>  Addr_t;
    typedef sc_uint<DataWidth>  Data_t;

    OSSS_REQUIRED(mem_if)
    {
      sc_out<Addr_t>   po_Addr;
      sc_in<Data_t>    pi_Data;
      sc_out<Data_t>   po_Data;
      sc_out<bool>     po_bEnable;
      sc_out<bool>     po_bR_W;
    };

    OSSS_GUARDED_METHOD_VOID(write,
                             OSSS_PARAMS(2, Addr_t, addr, Data_t, data),
                             true);

    OSSS_GUARDED_METHOD(Data_t,
                        read, OSSS_PARAMS(1, Addr_t, addr), true);
};
```

Listing 6.4: Declaration of a socket

```
#include "SharedMemAdapter.hh"

void
SharedMemAdapter::write(Addr_t addr, Data_t data)
{
  mem_if().po_bEnable.write(true);
```

```
7     mem_if().po_bR_W.write(false);
8     mem_if().po_Addr.write(addr);
9     mem_if().po_Data.write(data);
10    wait();
11    mem_if().po_bEnable.write(false);
12  }
13
14  SharedMemAdapter::Data_t
15  SharedMemAdapter::read(Addr_t addr) const
16  {
17    mem_if().po_bEnable.write(true);
18    mem_if().po_bR_W.write(true);
19    mem_if().po_Addr.write(addr);
20    wait(2);
21    mem_if().po_bEnable.write(false);
22    return mem_if().pi_Data.read();
23  }
```

Listing 6.5: Implementation of a socket

The implementation of the module resembles the implementation of a Shared Object. The main difference are the accesses to low level signals from within the Shared Object. Accesses to the signals of a socket interface are done the following way: `if_name.signal_name.read()` and `if_name.signal_name.write()`. By using this statements signals can be manipulated from within a Shared Object.

Instantiation of a socket

The instantiation of a socket is quite simple, as can be seen in listing 6.6. The socket variable is declared of the template type `osss_socket< >` and the class of the socket is passed as template parameter during declaration. In this case the socket type is a Shared Object. Every class used as a socket must contain an `OSSS_REQUIRED` section.

```
1   [...]
2  osss_socket<osss_shared<SharedMemAdapter> > m_SharedMem;
3   [...]
```

Listing 6.6: Instantiation of a socket

Binding signals to a socket

In this case the socket also contains a Shared Object, so the clients have to be bound to the socket. On the other hand the socket has a port interface that has to be connected to signals in the surrounding module. The binding of the Shared Object is not be subject of this section.

```
1   [...]
2    m_SharedMem.get_interface().po_Addr(ms_Addr);
3    m_SharedMem.get_interface().pi_Data(ms_rData);
4    m_SharedMem.get_interface().po_Data(ms_wData);
5    m_SharedMem.get_interface().po_bEnable(ms_bEnable);
6    m_SharedMem.get_interface().po_bR_W(ms_bR_W);
7   [...]
```

Listing 6.7: Binding socket signals

Binding socket signals to signals in the surrounding module is similar to the binding of port signals. The only difference is the use of the `get_interface()` method provided by the socket object. Listing 6.7 contains an example of how the socket signals are bound. Additionally the object plugged into the socket can be returned by invoking the `plugged_object()` method of the socket.

6.4.2.2 Restrictions

To summarize the sections above, sockets are subject to the following restrictions.

SO-AS-R1: The socket class must contain an `OSSS_REQUIRED` section.

SO-AS-R2: Accesses to socket signals from within the socket class must be done using `if_name().signal_name.read()` or `if_name().signal_name.write()` where `if_name` is the name of the socket interface, which is defined in the `OSSS_REQUIRED` section.

SO-AS-R3: The socket class must be a Shared Object.

SO-AS-R4: Socket variables must be declared of the `osss_socket< >` template type.

SO-AS-R5: Socket signals must be bound using the `get_interface()` method of the socket variable.

SO-AS-R6: The `plugged_object()` can be used to get the object plugged into a socket.

6.4.3 Software Task

Natively, SystemC does not support the modeling of software. Although it is very easy to write algorithms in a sequential and "untimed" or causal timed model, SystemC does not support a well defined synchronization between hardware and software models. The simulation time is managed by the SystemC kernel and can only be advanced by calling the `wait()` function. The execution of the statements between two successive wait statements does not affect the simulation time maintained by the kernel. Hence, for a proper synchronization of hardware and software components it is necessary to introduce a notion of software execution time.

One possibility to introduce execution times would be to use an explicit CPU model (e.g. based on an instruction set simulator) to execute the software. This approach has two main disadvantages. Firstly, the simulation performance of an instruction-level simulation is inferior to a native host code execution and secondly it complicates the introduction of an abstract communication mechanism between hardware and software. Therefore, we propose an approach based on the block-level annotation of execution times which overcomes these two disadvantages.

To overcome these limitations of SystemC a new class called `osss_software_task` is introduced. An OSSS Software Task is the counterpart to a `sc_module`. While an `sc_module` is a structural component specialized for the description of hardware, which is parallel by nature, and can contain an arbitrary number of (parallel) processes (`SC_METHOD`, `SC_CTHREAD` or `SC_THREAD`), an arbitrary number of `sc_modules` (hierarchical modules) and an arbitrary number of `sc_ports` for communicating with the "outside world", an `osss_software_task` is a structural component specialized for the description of sequential software. It only contains a single thread of control that is provided by a method called `main()` and implemented as `SC_CTHREAD`[7]. For the `osss_software_task` two predefined ports, a `clock_port` and a `reset_port`, both of type `sc_in<bool>` are defined. These ports have to be bound to the top-level's global clock and reset signals. Beside these predefined ports, the software task can contain an arbitrary number of ports of type `osss_port<osss_shared_if<`*IF*`> >`. These ports are used to communicate with Shared Objects (see Section 6.4.1). In contrast to `sc_module`s no nesting of `osss_software_task`s is allowed.

Before presenting the usage of software tasks by example, Table 6.2 compares the properties of `sc_module` and `osss_software_task`.

6.4.3.1 Declaration of a Software Task

Listing 6.8 shows the declaration of a software task in OSSS. For convenience the `OSSS_SOFTWARE_TASK` macro can be used instead of `class my_software_task : public osss::osss_software_task`. Similar to `SC_MODULE`s the default constructor can be written by using the `OSSS_SOFTWARE_CTOR` or `OSSS_SW_CTOR` macro. The method `main` is declared pure virtual in the base class `osss_software_task` and needs to be implemented by the user. This method represents the single thread of control for a software task.

[7] This is a technical restriction of the current OSSS implementation and should be changed to `SC_THREAD` in the future.

	sc_module	osss_software_task
Purpose	Structural element for the modeling of parallel hardware	Structural element for the modeling of sequential software
Class declaration macro	SC_MODULE(*class*)	OSSS_SW_TASK(*class*) OSSS_SOFTWARE_TASK(*class*)
Constructor macro	SC_CTOR(*class*)	OSSS_SW_CTOR(*class*) OSSS_SOFTWARE_CTOR(*class*)
Number of processes	0-N	1 (single thread of control)
Type of processes/ Notion of time	SC_METHOD / next_trigger()	-
	SC_CTHREAD / wait()	SC_CTHREAD / OSSS_EET(*time*)
	SC_THREAD / wait(*time*)	SC_THREAD / OSSS_EET(*time*)
Pre-defined ports	-	sc_in<bool> clock_port
	-	sc_in<bool> reset_port
Communication ports	sc_port<...> (0-N) osss_port<osss_shared_if<...> > (0-N)	osss_port<osss_shared_if<...> > (0-N)
Hierarchy/Nesting allowed	yes (arbitrary depth)	no

Table 6.2: Comparison between `sc_module` and `osss_software_task`

```
1  OSSS_SOFTWARE_TASK(my_software_task) {
2  public:
3
4    OSSS_SOFTWARE_CTOR(my_software_task) { }
5
6    // alternative constructor
7    my_software_task(...) : osss_software_task() {
8      /* put your software task constructor
9         code here */
10     [...]
11   }
12
13   virtual void main() {
14     /* put your software code here */
15     [...]
16   }
17 };
```

Listing 6.8: Declaration of a Software Task

6.4.3.2 Instantiation of a Software Task

Listing 6.9 shows the instantiation of `my_software_task` as declared in Listing 6.8. To “run” a software task, its pre-defined clock and reset ports need to be bound to the tob-level’s clock and reset signals. Please note that both ports `clock_port` and `reset_port` are inherited from class `osss_software_task`. That is why these ports are available in class `my_software_task` and need to be bound, although non of them has been declared in Listing 6.8.

```
1  #define OSSS_BLUE
2  #include <osss.h>
3  #include "my_software_task.h"
4
5  SC_MODULE(Top) {
6  public:
7
8    sc_in<bool> clk;
9    sc_in<bool> reset;
10
11   my_software_task* mt;
12
13   SC_CTOR(Top) {
14     mt = new my_software_task("my_software_task");
15     // perform binding of special clock and reset ports
```

```
16      mt->clock_port(clk);
17      mt->reset_port(reset);
18    }
19  };
```

Listing 6.9: Instantiation of a Software Task

6.4.3.3 Using EETs for specifying the software timing behaviour

In our approach we distinguish two types of execution times: the **E**stimated **E**xecution **T**ime (EET) and the **R**equired **E**xecution **T**ime (RET). The EET as shown in Listing 6.10 defines the execution time of the enclosed code block. These annotated times will only affect the simulation and do not have any synthesis semantics. In principle these times can automatically be obtained (e.g. through profiling on the target CPU) and back-annotated into the model.

In order to achieve a realistic simulation it is necessary to impose two constraints on the usage of EETs. Firstly, no communication with other modules must happen within an EET block and, secondly, there must be no code between the end of an EET block and a communication statement. The EETs lead to a more accurate timing behaviour than relying on synchronization through communication alone.

Listing 6.10 shows how to use `EET` blocks to specify the software timing behaviour. Besides the `OSSS_EET` macro we are using the `PRINT_MSG` macro which prints a kind of execution trace to the console. This macro does not influence the model execution, it just prints the current simulation time and call context (i.e. hierarchical module name, source code line and user-defined string). Have a look at Listing 6.11 to see what this message macro writes to the console during the execution of `task1`. Each line in Listing 6.11 corresponds to a call of the `PINT_MSG` macro. The `EXPECTED_TIME` macro is used to check whether the proper simulation time has passed. When calling `EXPECTED_TIME(sc_time(110.0, SC_NS))` it is expected that exactly 110.0 nanoseconds (ns) of simulation time has passed. If either more or less time has passed the macro writes an error to the console and quits the simulation. Like the `PRINT_MSG` macro the `EXPECTED_TIME` macro is some kind of assertion that does not influence the model execution either.

```
72  OSSS_SOFTWARE_TASK(task1) {
73  public:
74
75    OSSS_SOFTWARE_CTOR(task1) { }
76
77    void methodX() {
78      PRINT_MSG("Beginning methodX");
79      OSSS_EET(sc_time(3.0, SC_US)) {
80        /* do something else */
81      }
82      PRINT_MSG("Completed methodX");
83    }
84
85    virtual void main() {
86      OSSS_EET(sc_time(2.0, SC_US)) {
87        /* do something */
88        PRINT_MSG("Doing something");
89      }
90      PRINT_MSG("Communication with some other module");
91
92      EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
93                    sc_time(2.0, SC_US));
94
95      // Note: The execution time of the initialisation of i and
96      //       for checking the condition (at least the first
97      //       time) is neglected here
98      for (int i=0; i<3; ++i) {
99        OSSS_EET(sc_time(5.0, SC_US)) {
100         PRINT_MSG("For loop, iteration " << i);
101
102         if (i%2 == 0) {
103           // will be called for i==0 and i==2
104           methodX();
```

```
105          }
106        }
107
108        EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
109                      sc_time(2.0, SC_US)+
110                      sc_time(5.0, SC_US)*static_cast<double>(i+1)+
111                      sc_time(3.0, SC_US)*((i==2) ? 2.0 : 1.0)
112                     );
113      }
114    }
115  };
```

Listing 6.10: Usage of EETs in a Software Task

Listing 6.11 shows the console output after "running" the above software task. A clock period of 10.0 ns is used. Since the simulation starts with 10 synchronous reset cycles the first time stamp occurs at 110 ns.

```
110 ns   : top.task1(line: 94)  Doing something
2110 ns  : top.task1(line: 96)  Communication with some other module
2110 ns  : top.task1(line: 107) For loop, iteration 0
2110 ns  : top.task1(line: 83)  Beginning methodX
5110 ns  : top.task1(line: 87)  Completed methodX
10110 ns : top.task1(line: 107) For loop, iteration 1
15110 ns : top.task1(line: 107) For loop, iteration 2
15110 ns : top.task1(line: 83)  Beginning methodX
18110 ns : top.task1(line: 87)  Completed methodX
```

Listing 6.11: Console output after running the Software Task from Listing 6.10 (Clock period is 10.0 ns, number of reset cycles is 10)

6.4.3.4 Using EETs and RETs for checking timing consistencies of Software Tasks

Syntactically the specification of RETs is almost identical to the specification of EETs, but the simulation semantics is different. The RET results in a piece of code which will not consume any simulation time at all. It can be considered as a timing constraint on the contained EET blocks and optional calls to the outside world (e.g. a Shared Object implemented in hardware).

It is also possible to mix and nest EETs and RETs. Doing so will allow for finding RET violations during the simulation. For instance, if an RET block of 5 ms contains a loop whose body has an EET of 1 ms per iteration and it performs more than 5 iterations in a simulation run, the RET block will report an error.

Listing 6.12 shows the usage of `EET` and `RET` blocks for checking timing consistencies of software tasks. In this example all `OSSS_EETs` are enclosed by `OSSS_RET` (**R**equired **E**xecution **T**ime) blocks. They report a timing violation when the amount of time that is passed inside an `RET` block is bigger than specified. In this example the `RET` in line 181 is intentionally violated by the inner `EET` block that is executed in a loop for three times. Please note that the timing violation is reported not until the `RET` block is left.

```
124  OSSS_SOFTWARE_TASK(task2) {
125  public:
126
127    OSSS_SW_CTOR(task2) { }
128
129    void methodY() {
130      OSSS_RET(sc_time(6.0, SC_US)) {
131
132        OSSS_EET(sc_time(4.0, SC_US)) {
133
134        }
135      }
136    }
137
138    virtual void main() {
139      PRINT_MSG("Beginning time critical calculation");
140      OSSS_RET(sc_time(10.0, SC_US)) {
141
```

```
142       OSSS_RET(sc_time(4.0, SC_US)) {
143
144         PRINT_MSG("Beginning time critical sub-calculation 1");
145         OSSS_EET(sc_time(2.0, SC_US)){
146           /* do something */
147         }
148         PRINT_MSG("Completed time critical sub-calculation 1");
149
150         EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
151                       sc_time(2.0, SC_US));
152
153       }
154
155       EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
156                     sc_time(2.0, SC_US));
157
158       PRINT_MSG("Beginning time critical sub-calculation 2");
159       OSSS_EET(sc_time(2.0, SC_US)) {
160         /* do something */
161       }
162       PRINT_MSG("Completed time critical sub-calculation 2");
163
164       EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
165                     sc_time(4.0, SC_US));
166
167     }
168
169     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
170                   sc_time(4.0, SC_US));
171
172     PRINT_MSG("Completed time critical calculation");
173
174     PRINT_MSG("Beginning time critical calculation 2 (which will fail)");
175     OSSS_RET(sc_time(3.0, SC_US)) {
176
177       for (int i=0; i<3; ++i) {
178         OSSS_EET(sc_time(2.0, SC_US)) {
179           PRINT_MSG("For loop, iteration " << i);
180         }
181         EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
182                       sc_time(4.0, SC_US)+
183                       sc_time(2.0, SC_US)*static_cast<double>(i+1));
184
185       }
186     }
187
188     // The previous RET is intentionally violated by inner EETs.
189     // Hence we expect now == 10.0 us instead of 7.0 us
190     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
191                   sc_time(10.0, SC_US));
192
193     PRINT_MSG("Completed time critical calculation 2");
194
195     PRINT_MSG("Beginning time critical calculation 3 (which is
        inconsistently constrained)");
196     OSSS_RET(sc_time(5.0, SC_US)) {
197       methodY();
198     }
199
200     // The previous RET is intentionally violated by an inner RET.
201     // Hence we expect now == 10.0 us instead of 7.0 us
202     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
203                   sc_time(14.0, SC_US));
204
205     PRINT_MSG("Completed time critical calculation 3");
206   }
207 };
```

Listing 6.12: Usage of EETs and RETs in a Software Task

Listing 6.13 shows the console output after "runnig" the above software task. The `RET`

violation in line 181 is reported as expected.

```
110 ns   : top.task2(line: 146)  Beginning time critical calculation
110 ns   : top.task2(line: 151)  Beginning time critical sub−calculation 1
2110 ns  : top.task2(line: 155)  Completed time critical sub−calculation 1
2110 ns  : top.task2(line: 165)  Beginning time critical sub−calculation 2
4110 ns  : top.task2(line: 169)  Completed time critical sub−calculation 2
4110 ns  : top.task2(line: 179)  Completed time critical calculation
4110 ns  : top.task2(line: 181)  Beginning time critical calculation 2
                                 (which will fail)
4110 ns  : top.task2(line: 187)  For loop, iteration 0
6110 ns  : top.task2(line: 187)  For loop, iteration 1
8110 ns  : top.task2(line: 187)  For loop, iteration 2
OSSS_RET violation [top.task2, sw_timing.cpp:182, created : 4110 ns] :
                   duration: 3 us, deadline: 7110 ns, now: 10110 ns
10110 ns : top.task2(line: 201)  Completed time critical calculation 2
10110 ns : top.task2(line: 203)  Beginning time critical calculation 3
                                 (which is inconsistently constrained)
14110 ns : top.task2(line: 213)  Completed time critical calculation 3
```

Listing 6.13: Console output after running the Software Task from Listing 6.12 (Clock period is 10.0 ns, number of reset cycles is 10)

6.4.3.5 Restrictions when using Software Tasks

Software Tasks have the following restrictions:

SW-R1: Must be derived from class `osss_software_task` or use the `OSSS_SOFTWARE_TASK(...)` or `OSSS_SW_TASK(...)` macros.

SW-R2: Must implement the pure virtual `main()` routine. The user-defined software behavior is implemented in this routine only. No other processes (`SC_METHOD`, `SC_CTHREAD` or `SC_THREAD`) are allowed inside a software task.

SW-R3: A software task may not contain any other software task or module.

SW-R4: Pre-defined ports `clock_port` and `reset_port` need to be bound to the global clock. No other signal ports are allowed for software tasks.

SW-R5: Communication is realized via `osss_port`s bound to Shared Objects only. No other communication ports are allowed.

SW-R6: On other timing annotation than `OSSS_EET(...)` blocks are allowed. A call of the SystemC `wait(...)` function is not allowed. For restrictions of software task timing annotations see below.

For timing annotations within software tasks using Estimated Execution Time (EET) blocks the following restrictions apply:

EEE-R1: EETs are C++ compound statements (commonly called “blocks”) and may only be used within the rules of C++.

EET-R2: EET blocks may not be nested (i.e. an EET block may not contain any other EET block).

EET-R3: EET blocks may not overlap (i.e. no new EET block is allowed to begin before the currently active EET block has not been closed).

EET-R4: EET blocks may not contain any Shared Object service calls on `osss_port`s.

For timing timing checks/assertions within software tasks using Required Execution Time (RET) blocks the following restrictions apply:

RET-R1: RETs are C++ compound statements (commonly called “blocks”) and may only be used within the rules of C++.

RET-R2: RET blocks may not overlap (i.e. no new RET block is allowed to begin before the currently active RET block has not been closed).

6.4.4 Hardware/Software Communication

Listing 6.14 shows the producer to be implemented in software. To implement the producer as a software task the Producer class has to be derived from the `osss_software_task`. The communication of a software task with Shared Objects is performed by the usage of specialized OSSS-Ports. The `osss_port` is derived from the SystemC `sc_port` and is bound to the instance of the Shared Object, see Listing 6.2. The `osss_shared_if` class implements a Shared Object interface class used as a base class for the Shared Object class (`osss_shared`). Thus, the interface of the `osss_port` has to be of type `osss_shared_if` to connect the `osss_port` of the software task to a Shared Object. Furthermore, the interface of the object type of the Shared Object has to be specified as interface of the `osss_shared_if`. In Listing 6.14 the interface of the object type that is implemented as a Shared Object is `FIFO_if`. Thus the output port of the Producer class is of type `osss_port<osss_shared_if<FIFO_if> >`.

The `FIFO` in the producer/consumer example is specified to store items of type `Packet`. The implementation of the `FIFO` is explained in more detail in Section 6.4.1. The methods inside of the FIFO object are called from the software task by the `operator->()` on the `osss_port`. In the example the `put` method is called on the output port.

```
1  class Producer : public osss_software_task {
2  public:
3
4    // connection to the shared object
5    osss_port<osss_shared_if< FIFO_if<Packet> > > output;
6
7    // runs only once in the beginning
8    OSSS_SW_CTOR(Producer) { }
9
10   // has to override the virtual main()
11   void main() {
12     Packet p;
13     while(true) {
14       OSSS_EET(sc_time(50.0, SC_NS)) {
15         /* some calculations that take approximately
16            50.0 nano seconds */
17       }
18
19       // communication with the "outside world"
20       output->put(p);
21
22       OSSS_EET(sc_time(10.0, SC_NS)) {
23         /* some calculations that take approximately
24            10.0 nano seconds */
25       }
26     }
27   }
28 };
```

Listing 6.14: OSSS-Software-Task with annotated Estimated Execution Times

Figure 6.17 illustrates the usage of EETs and RETs in the producer Software Task of the producer/consumer example. Listing 6.15 has the same block structure using EET and RET annotation as shown in Figure 6.17. The behaviour of this software task is to generate data of type `Packet` and to write them to a FIFO Shared Object. Until now we will only take a look on the block structure and the EET and RET annotations. The body of the infinite while loop (line 13) in the main process is constrained by an RET of 2000.0 nanoseconds (line 14). The following for loop (line 18) initializes the `Packet` object and assigns a dummy payload. Since communication with Shared Objects can not be inside EET blocks the call of the `put` method on the `output` port (line 21) is not within the packet initialization block. The same rule has been applied to the annotation of the following if condition (line 22); the else branch (line 28) contains a call to a Shared Object (line 34) and thus cannot be enclosed by an EET block around the entire if-statement.

```
1 OSSS_SW_TASK(Producer) {
2   osss_port<osss_shared_if<FIFO_put_if<Packet> > > output;
3
```

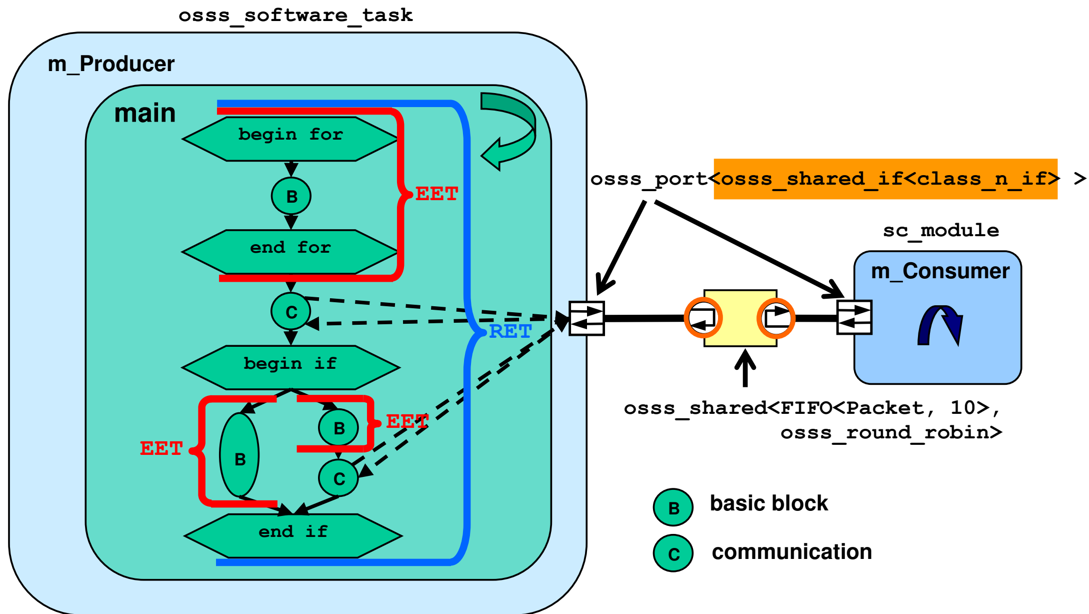


Figure 6.17: EET statements inside of the `main()` method of a software task [44]

```
4    OSSS_SW_CTOR(Producer) : output("output") {}
5
6   protected:
7    virtual void main() {
8      const unsigned char source_addr = 42;
9      unsigned char target_addr = 0;
10     unsigned int offset = 0;
11     Packet p;
12
13     while(true) {
14       OSSS_RET(sc_time(2000.0, SC_NS)) {
15         OSSS_EET(sc_time(120.0, SC_NS)) {
16           p.set_source_addr(source_addr);
17           p.set_target_addr(target_addr);
18           for(unsigned int i = 0; i<p.get_payload_size(); ++i)
19             p.set_payload(i, i+offset);
20         }
21         output->put(p);
22         if (target_addr >= 10) {
23           OSSS_EET(sc_time(10.0, SC_NS)) {
24             target_addr = 0;
25             offset  = 0;
26           }
27         }
28         else {
29           OSSS_EET(sc_time(30.0, SC_NS)) {
30             target_addr += 1;
31             offset  += 10;
32             p.set_target_addr(target_addr);
33           }
34           output->put(p);
35         }
36       }
37     }
38   }
39 };
```
Listing 6.15: Producer Software Task

Listing 6.16 shows the signature of the `Packet` class, which will be used in the following examples. It contains data members for a source and a target address and a payload of 10 bytes. To provide the concept of encapsulation the `Packet` class has several access methods to

its protected data members.

```
1  class Packet {
2   public:
3
4    Packet();
5
6    unsigned char get_source_addr() const;
7    void          set_source_addr(unsigned char addr);
8
9    unsigned char get_target_addr() const;
10   void          set_target_addr(unsigned char addr);
11
12   unsigned char get_payload(unsigned int index) const;
13   void          set_payload(unsigned int index,
14                             unsigned char data);
15
16   unsigned int  get_payload_size() const;
17
18  protected:
19   unsigned char  m_source_addr;
20   unsigned char  m_target_addr;
21   unsigned char  m_payload[10];
22 };
```

Listing 6.16: Signature of the `Packet` class

6.4.5 Hardware Module

Hardware on the *Application Layer* is described by the OSSS hardware subset which is basically the synthesizable SystemC subset, for mode details see Chapter F. A hardware module is an `SC_MODULE` with `SC_CTHREAD` and/or `SC_METHOD` processes which implement the behaviour.

Ports are used to communicate with other components: SystemC signal ports are used to communicate directly with other hardware modules. OSSS ports are used to establish the communication with Shared Objects.

The consumer is an `sc_module` implementing a single clocked process, which calls the `get` method on its input port continuously (line 16). The `get` method called on the local port is redirected to a call of the guarded method implemented in the `FIFO` class, because the input port is bound to the buffer Shared Object. This abstract communication mechanism is uniform for SW Tasks and HW modules. It hides the details of the Shared Object's communication protocol involving the scheduling and guard evaluation. Furthermore, it enables easy replacement, addition and removal of software tasks and hardware modules without changing the communication and synchronization between them.

```
1  SC_MODULE(Consumer) {
2    sc_in<bool> clk, reset;
3
4    osss_port<osss_shared_if<FIFO_get_if<Packet> > > input;
5
6    SC_CTOR(Consumer) : input("input") {
7      SC_CTHREAD(cons_process, clk.pos());
8      reset_signal_is(reset, true);
9    }
10
11  protected:
12   void cons_process() {
13     Packet p;
14     while(true) {
15       wait();
16       p = input->get();
17     }
18   }
19 };
```

Listing 6.17: Consumer HW module implementation

6.5 Virtual Target Architecture Layer[8]

This layer provides architecture building blocks to assemble an configure a System on Chip architecture. These building blocks are software processors, memories, and (user-defined) hardware blocks. For the interconnection of these blocks different communication channels, like buses, crossbar switches or point-to-point connections are available. All architecture building blocks are stored in a hierarchical *Architecture Class Library* that can be extended by user-defined architecture elements.

6.5.1 Architecture Class Library

Figure 6.18 shows the building blocks of the *Virtual Target Architecture* organized as a class hierarchy. The components shown are supported by the OSSS synthesis flow and can be used to build a synthesizable *Virtual Target Architecture*. The supported target architecture can be assembled from a subset of the Xilinx IP core library available in the Xilinx EDK (**E**mbedded **D**evelopment **K**it) [239] and the Xilinx ISE (**I**ntegrated **S**ynthesis **E**nvironment) [238]. Figure 6.18 only presents a set of selected architecture building blocks used in the following examples and evaluation of this work. The architecture class library can be extended by more Xilinx, other vendor and custom components[9].

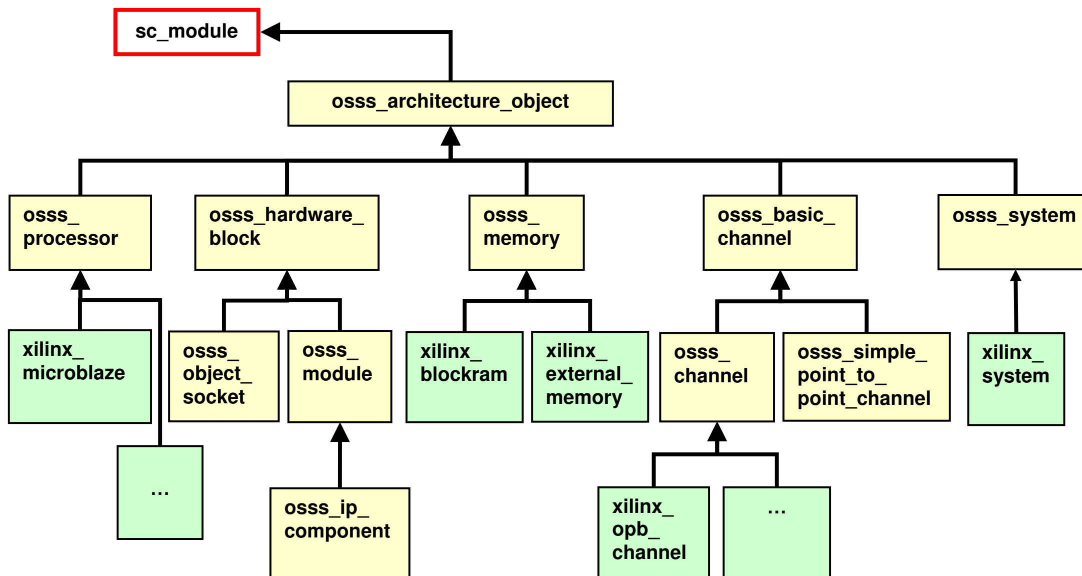


Figure 6.18: Sample of the OSSS Architecture Class Library [44]

All architecture building blocks in Figure 6.18 with a `xilinx_` prefix are wrapper classes for configurable platform IP components provided by Xilinx. More information about these components can be found in Section 7.4.

The other architecture building blocks are user-defined components: The `osss_hardware_block` is a base class for user-defined modules and OSSS Object Sockets. The `osss_module` is a specialization of an `sc_module` and adds a mandatory clock and reset port to assure that all processes are driven by a global clock and reset signal. There is no semantic difference between the `sc_module` and the `osss_module`. The `osss_ip_component` is just a wrapper for RT level IP components.

The OSSS-Channel [77, 78] is a concept to model on-chip the communication independently from RMI protocol and the behavior. It can be used for a cycle accurate specification of a physical channel model, like a bus or a custom designed point-to-point channel. More details are given in Section 6.5.3.

[8] This section is based on own previous work [44].
[9] This is denoted by the blocks labeled with "...") is but not further described in this work.

The OSSS-Memory class is used for the explicit specification of memories in the Target Architecture. In Xilinx FPGAs these dedicated memories can be either internal Block-RAM or external memory, like SRAM, DRAM or Flash. The `osss_system` and the specialized `xilinx_system` are top-level modules which represent the SoC boundary. All ports of the `xilinx_system` are mapped to FPGA pins.

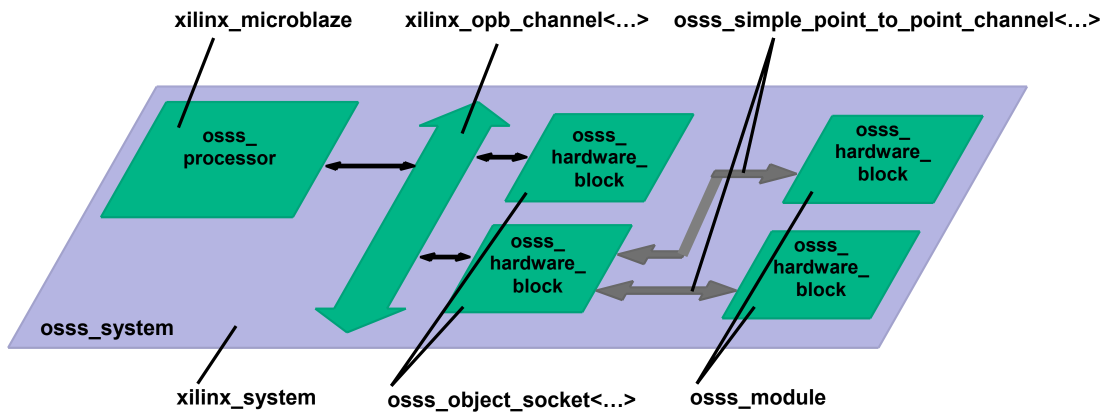


Figure 6.19: Example of a *Virtual Target Architecture* with a single processor and user-defined hardware [44]

Figure 6.19 shows an example of a *Virtual Target Architecture* composed of different OSSS *Architecture Objects*. It includes a single Xilinx MicroBlaze™processor block connected to a Xilinx On-Chip Peripheral Bus (OPB) [151, 105] as bus master. Two OSSS *Object Sockets* are connected to the OPB as slave components. The lower OSSS Object Socket is connected with two user-defined hardware-blocks by a point-to-point connection.

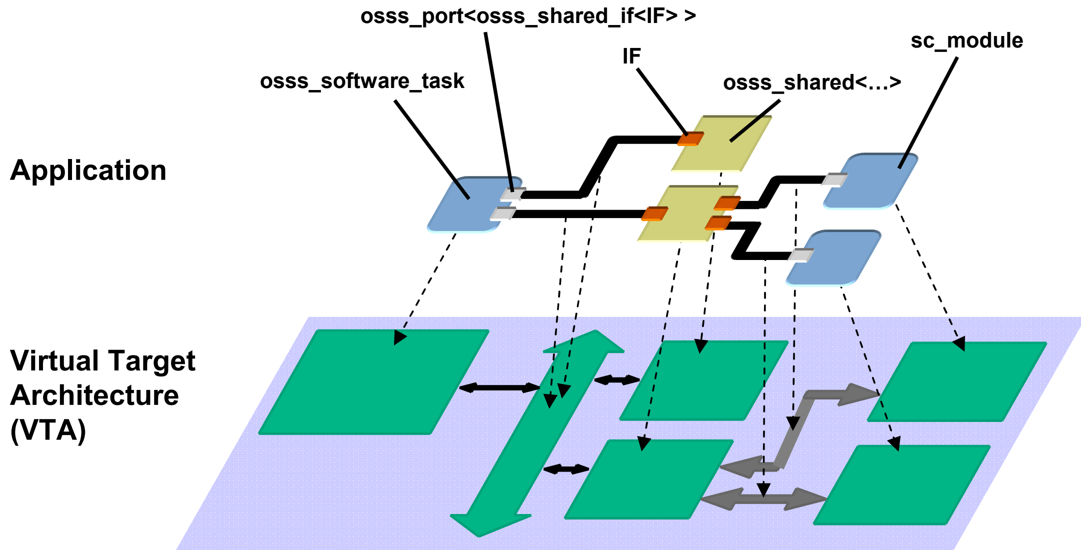
6.5.2 Remote Method Invocation

The general meaning of the term *Remote Method Invocation* (RMI) is the call of a method (function or procedure) of an object that is not directly accessible to the caller. I.e. it is not accessible in the instruction memory of the local processor executing certain software. The remote object has to be physically accessible through a communication network. On the *Virtual Target Architecture Layer* the communication network between software tasks, hardware modules, and Shared Objects is modeled by OSSS-Channels. Thus, RMI in the context of OSSS represents a method call from a port of a hardware module or software task to an interface of a Shared Object through an OSSS-Channel.

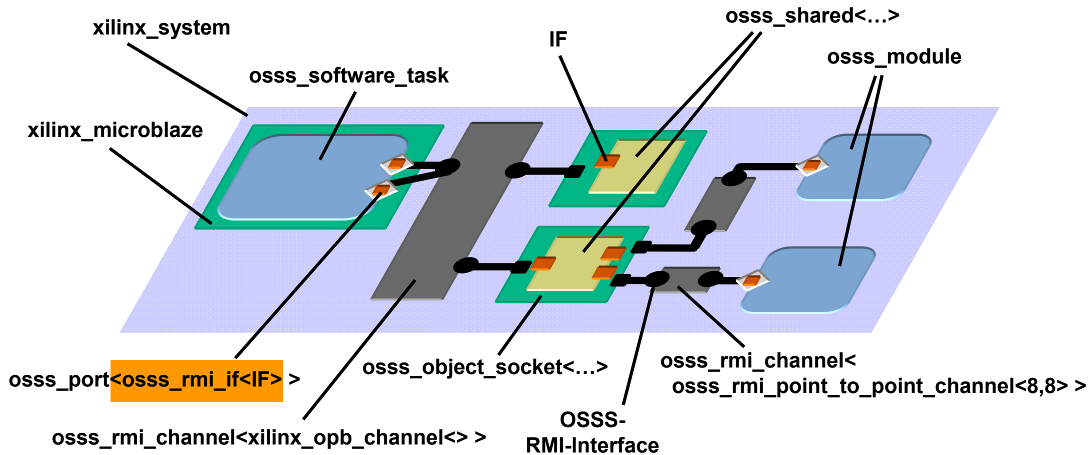
The subsequent sections are organized as follows: We start with the description of the general concept behind RMI. This is followed by the presentation of the OSSS-RMI protocol stack. Afterwards, we show the particular steps of the OSSS design flow that need to be performed in order to map an application to a specific target architecture. For a better illustration we perform these steps for the simple consumer/producer example from Section 6.4. After the presentation of the mapping steps we demonstrate the flexible communication refinement provided by the OSSS methodology. It enables a simulative architecture exploration, demonstrated with the producer/consumer example. Finally, we are going to have a look at the physical layer of the OSSS-RMI protocol. The OSSS-Channel concept provides building blocks for the specification of synthesizable and cycle accurate communication channels. These channels establish the signal level communication and transform method based into a signal based communication.

6.5.2.1 The General Concept

Figure 6.20b shows the results of the mapping and communication refinement applied to the example presented in Figure 6.20a. All communication links from the *Application Layer* have been mapped to `osss_rmi_channel<...>` containers. They serve as wrappers for the OSSS-Channels that implement the physical structure (bus, point-to-point, or crossbar) and the



(a) Application to VTA mapping



(b) Result of the Application to VTA mapping

Figure 6.20: Communicating processes mapped on *Virtual Target Architecture* [44]

behavior of the communication protocol. The purpose of the RMI-Channel is the provision of a generic OSSS-RMI interface and the translation of the OSSS-RMI protocol to the generic OSSS-Channel protocol.

The `osss_software_task` has been mapped on a Xilinx MicroBlaze processor that is connected to an OPB, implemented by an OSSS-Channel (`xilinx_opb_channel`). Since we do not want to manually refine all method calls between the Software Task and the Shared Objects, we need to wrap the OPB channel by an OSSS-RMI container (`osss_rmi_channel<...>`). The same wrapping needs to be performed for the point-to-point connection (`osss_rmi_point_to_point_channel<...>`) from the `osss_modules` to the Shared Object.

As already mentioned above, each Shared Object needs to be wrapped by an `osss_object_socket<...>` container. This socket provides a binding mechanism to the `osss_rmi_channel<...>` container. Moreover, it performs the OSSS-RMI protocol and finally calls the remotely requested service on the Shared Object. Thus serving as a virtual "local client" to the Shared Object.

For calling a remote method from inside of a software task or a hardware module, their communication ports need to be prepared for RMI. This is a rather technical implication. Like in all known RMI concepts a local stub or proxy for accessing the remote object has to be

generated. When performing a method call on a local stub the RMI protocol becomes initiated. This implies a sequence of the following operations:

1. serialization of the remote method arguments, performed at the stub (`osss_rmi_if<IF>`),
2. submission of the client ID, method ID and the serialized arguments, also performed at the stub,
3. reception of the client ID, method ID and the serialized parameters at the remote object socket (`osss_object_socket<...>`),
4. de-serialization of the parameters at the remote object socket,
5. call of the method by assigning the de-serialized parameters to the method of the remote object (`osss_object_socket<osss_shared<...> >`).

When a method with non-void return parameter has been called, the same protocol is performed on the way back from the remote object to the caller. This caller stub is denoted as `osss_rmi_if<...>`. For RMI communication each OSSS-Port needs to be equipped with this stub. It can be derived automatically from the abstract interface class of the connected Shared Object, as describe in Section 6.6.

6.5.2.2 RMI protocol stack

After presenting the general RMI concept we will go into more detail concerning the OSSS-RMI. Figure 6.21 shows the OSSS-RMI protocol stack and gives an overview of the building blocks and their interfaces.

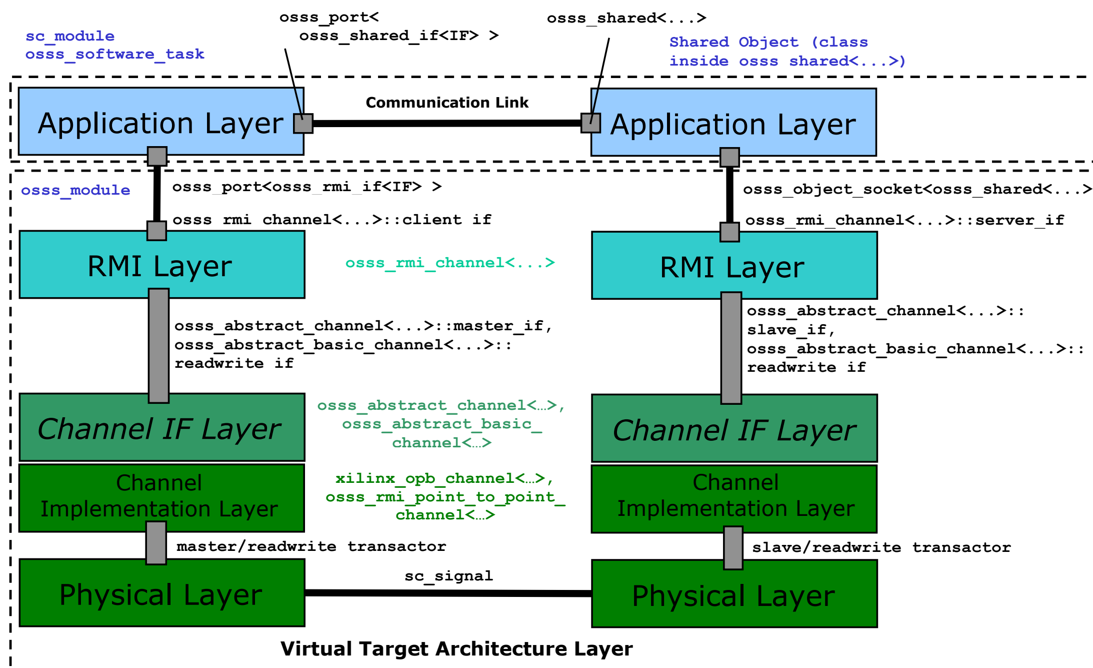


Figure 6.21: The OSSS-RMI protocol stack [44]

On the Application Layer OSSS provides three building blocks: `sc_module` for describing custom hardware, `osss_software_task` for describing software, and `osss_shared_object<...>` for describing shared resources, inter process communication, and method interfaces for hardware components. Communication between `sc_modules` or `osss_software_tasks` and Shared Objects is described by an `osss_port<osss_shared_if<IF> >` to `osss_shared<...>` binding. This kind of port to interface binding on the Application Layer creates a direct communication

link between the module or the software task and the Shared Object. This kind of communication modeling is used on the Application Layer as shown in the uppermost part of Figure 6.21.

As shown in the producer/consumer example a communication link on the Application Layer can be refined by an OSSS-Channel. Since each OSSS-Channel describes a synthesizable signal based communication, the method call on the Application Layer has to be translated into a signal based communication and vice versa. This translation is performed by the OSSS-RMI protocol.

In the following sections the different layers and interfaces of the OSSS-RMI protocol stack are introduced. The concept and the implementation of OSSS-Channels is explained in Section 6.5.3.

The osss_rmi_channel<...>::client_if

The osss_rmi_channel<...> container provides two interfaces:

- the osss_rmi_channel<...>::client_if interface is used for communication between the osss_port<osss_rmi_if<IF> > and the osss_rmi_channel<...> and
- the osss_rmi_channel<...>::server_if interface is used for communication between the osss_object_socket<...> and the osss_rmi_channel<...>.

```
1 virtual OSSS_Archive_t call_function(OSSS_ClientID_t clientID,
2                                      OSSS_ObjectID_t objectID,
3                                      OSSS_MethodID_t methodID,
4                                      OSSS_Archive_t archive) = 0;
5
6 virtual void call_procedure(OSSS_ClientID_t clientID,
7                             OSSS_ObjectID_t objectID,
8                             OSSS_MethodID_t methodID,
9                             OSSS_Archive_t archive) = 0;
```

Listing 6.18: The osss_rmi_channel<...>::client_if

Listing 6.18 shows the osss_rmi_channel<...>::client_if that is used by the osss_port<osss_rmi_if<IF> > to initiate the communication. The osss_port<osss_rmi_if<IF> >, or more precisely the stub (i.e. the osss_rmi_if<IF>), translates the invocation of a remote method with a non void return parameter to a call of the call_function(...) method. If the remote method invocation has a void return parameter the call_procedure() method is invoked. The clientID, objectID, methodID (all typedefs to unsigned int) will be described below. The OSSS_Archive_t is a typedef to the type osss_serialisable_archive which will also be described in the following.

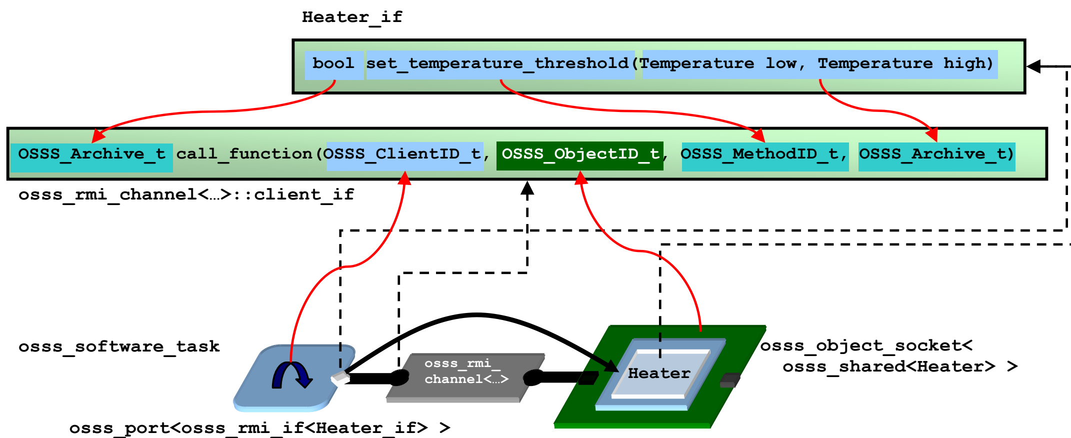


Figure 6.22: Usage of the osss_rmi_channel<...>::client_if when performing an RMI [44]

Figure 6.22 illustrates the usage of the osss_rmi_channel<...>::client_if when performing an RMI with a non void return parameter. In this example a software

task controls a heater implemented as a Shared Object. Assuming the software task calls the `set_temperature_threshold(...)` method on its OSSS-Port. Both the `osss_rmi_if<Heater_if>` and the heater class inside of the Shared Object implement the `Heater_if` interface. Thus, the call to the local `set_temperature_threshold(...)` method of the `osss_rmi_if<Heater_if>` stub has to be transformed to a call of the "real" implementation inside of the heater class of the Shared Object. The first step in each RMI call is the collection of all necessary information for performing the call. This information consists of the unique identification of the process performing the method call (`OSSS_ClientID_t`), the unique identification of the object implementing the called method (`OSSS_ObjectID_t`), the unique identification of the method itself (`OSSS_MethodID_t`) and their parameters. After collecting this information the `call_function(...)` or `call_procedure(...)` method of the `osss_rmi_channel<...>::client_if` is invoked.

The client ID can be extracted from the process ID and the object ID can be extracted from the `osss_object_socket<...>`. These IDs need to be unique with respect to the connected RMI-Channel only. The method ID is extracted from the name of the method inside the `osss_rmi_if<Heater_if>` stub. It only needs to be unique with respect to the class implementing it. All parameters of the method need to be serialized into a single bit-vector of type `OSSS_Archive_t`. As stated earlier in this document, all parameter types of a method need to be serializable (i.e. derived from `osss_serialisable_object`). This is an important pre-condition because serializable archives of type `OSSS_Archive_t` can only be constructed out of `osss_serialisable_objects`.

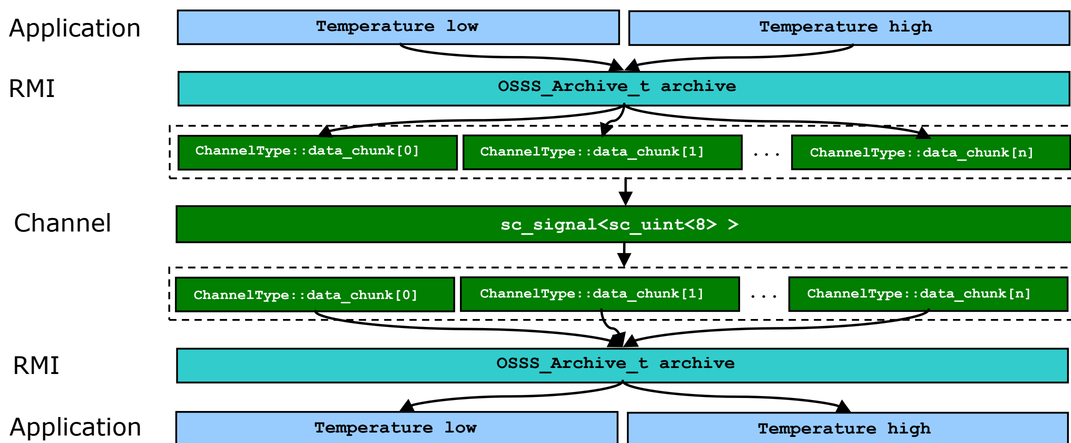


Figure 6.23: Serialization and de-serialization of method parameters when performing an RMI [44]

Figure 6.23 shows the serialization and de-serialization of method parameters necessary for performing an RMI. In our example above the two method parameters of type Temperature need to be serializable objects. On the RMI Layer they are both added to a serializable archive. The Channel Layer transforms the serializable archive to data chunks of the width of the signal interface of the channel (e.g. `sc_uint<8>`) and transmits it through the channel by using its specific protocol (e.g. using the `xilinx_opb_channel<...>` a burst transfer is initiated if the number of data chunks is greater than eight). On the other side of the channel (i.e. at the slave interface) data chunks are used to rebuild the serializable archive. Afterwards, the serializable objects of type Temperature can be extracted from the serializable archive.

Figure 6.24 shows the simplified RMI protocol state machine of the `osss_rmi_channel<...>::client_if`. When either the `call_function(...)` or the `call_procedure(...)` method is called on the `osss_rmi_channel<...>::client_if` the following states are traversed:

- `request_for_method`: The request for the call of a certain guarded method of a Shared Object is initiated. The method ID of the requested method is send to the appropriate

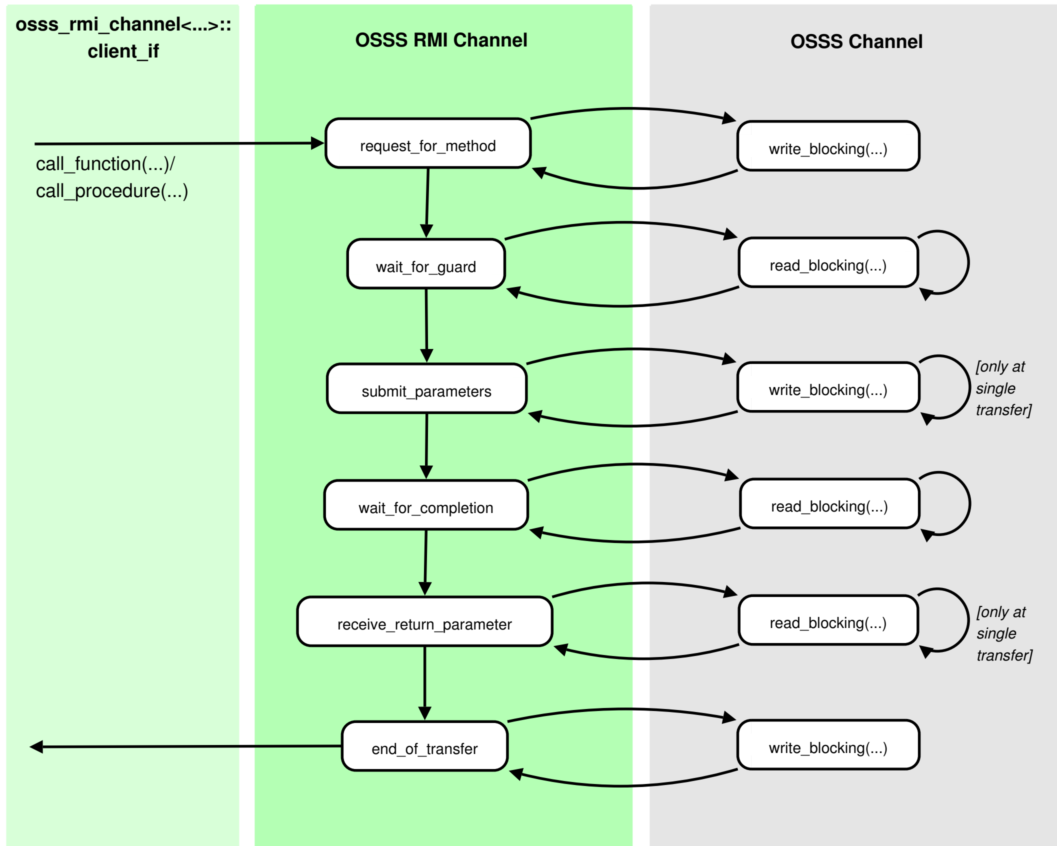


Figure 6.24: RMI protocol state machine of the `osss_rmi_channel<...>::client_if` [44]

`osss_object_socket<...>` by a call to the `write_blocking(...)` method of the OSSS-Channel.

- `wait_for_guard`: The `osss_object_socket<...>` is polled by calling the `read_blocking(...)` method on the OSSS-Channel. If the `read_blocking(...)` method returns true the access to the Shared Objects guarded method has been granted. If it return false the polling is continued.
- `submit_parameters`: The parameters of the called method are transferred through the OSSS-Channel by calling the `write_blocking(...)` method.
- `wait_for_completion`: The `osss_object_socket<...>` is polled by calling the `read_blocking(...)` method on the OSSS-Channel. If the `read_blocking(...)` method returns true the requested method call on the Shared Objects has been completed. If it return false the polling is continued.
- `receive_return_parameter`: The return parameter of the called method is transferred through the OSSS-Channel by calling the `read_blocking(...)` method.
- `end_of_transfer`: The method ID of the requested method is submitted to the corresponding `osss_object_socket<...>` by a call to the `write_blocking(...)` method of the OSSS-Channel. This last protocol phase quits the method call on the Shared Object.

The `osss_rmi_channel<...>::server_if`

For the communication between the `osss_object_socket<osss_shared<...> >` and the `osss_rmi_channel<...>` the `osss_rmi_channel<...>::server_if` is used. Listing 6.19 shows the services provided by the `osss_rmi_channel<...>::server_if` interface.

```
1  virtual void listen_for_action(OSSS_ClientID_t &clientID,
2                                 OSSS_ObjectID_t &objectID,
3                                 OSSS_MethodID_t &methodID,
4                                 OSSS_MethodID_Record_t* methodID_record,
5                                 bool initial = true) = 0;
6
7  virtual void wait_for_guard(OSSS_ClientID_t clientID,
8                              bool is_busy = true) = 0;
9
10 virtual bool receive_in_params(OSSS_Archive_t &arch) = 0;
11
12 virtual bool provide_return_params(OSSS_ClientID_t clientID,
13                                    OSSS_Archive_t &arch) = 0;
14
15 virtual void return_params_idle(OSSS_ClientID_t clientID,
16                                 bool is_busy = true) = 0;
```

Listing 6.19: The `osss_rmi_channel<...>::server_if` interface

To understand how these interface methods are used we first will have a closer look at the internal organization of the `osss_object_socket<osss_shared<...> >`. As shown in Figure 6.25, the `osss_object_socket<osss_shared<...> >` mainly consists of a Shared Object, a memory block used to store method parameters and at least one `osss_object_socket_port`. The `osss_object_socket_port` is composed of an `osss_port<...>`, an RMI protocol process, a memory called `methodID_record` and an *invoke method* process.

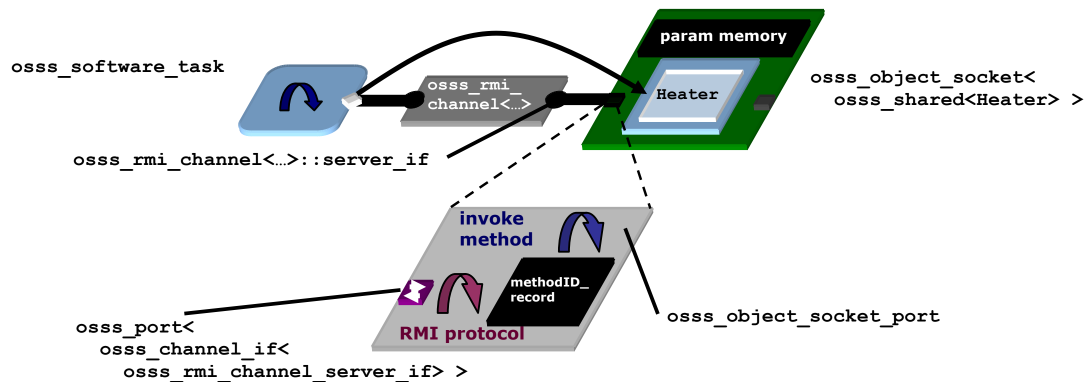


Figure 6.25: Organisation of the `osss_object_socket<osss_shared<...> >` container [44]

The `osss_port<osss_channel_if<...> >` is bound to an `osss_rmi_channel<...>` and is used to access the `osss_rmi_channel<...>::server_if` that is implemented inside the RMI-Channel. The RMI protocol process uses this port and performs the communication with the RMI-Channel. The `methodID_record` is used by the RMI protocol process (shown in Figure 6.26 as state machine) to store its state and to exchange information with the *invoke method* process. The *invoke method* process is used for the communication with the Shared Object inside the `osss_object_socket<...>`. It is some kind of "virtual client" to the Shared Object because it represents all client processes (inside `osss_modules` or `osss_software_tasks`) attached to the other side of the RMI-Channel that are registered by this instance of the Shared Object. The invoke method process checks for a valid guard in order to execute the requested method. If the guard is valid it executes the method of the Shared Object.

To get a better understanding of the RMI protocol executed inside the `osss_object_socket_port` we will explain the state machine shown in Figure 6.26. The syntax of the transition labels is:

Boolean Condition / Action

Multiple actions are separated by commas. For the sake of clarity conditions are written in black letters while actions are written in red letters. Time passes only when performing a call to

the `osss_port<osss_channel_if<...> >` denoted by `port->methodName(...)`, `methodName` is one of the methods of the `osss_rmi_channel<...>::server_if` interface.

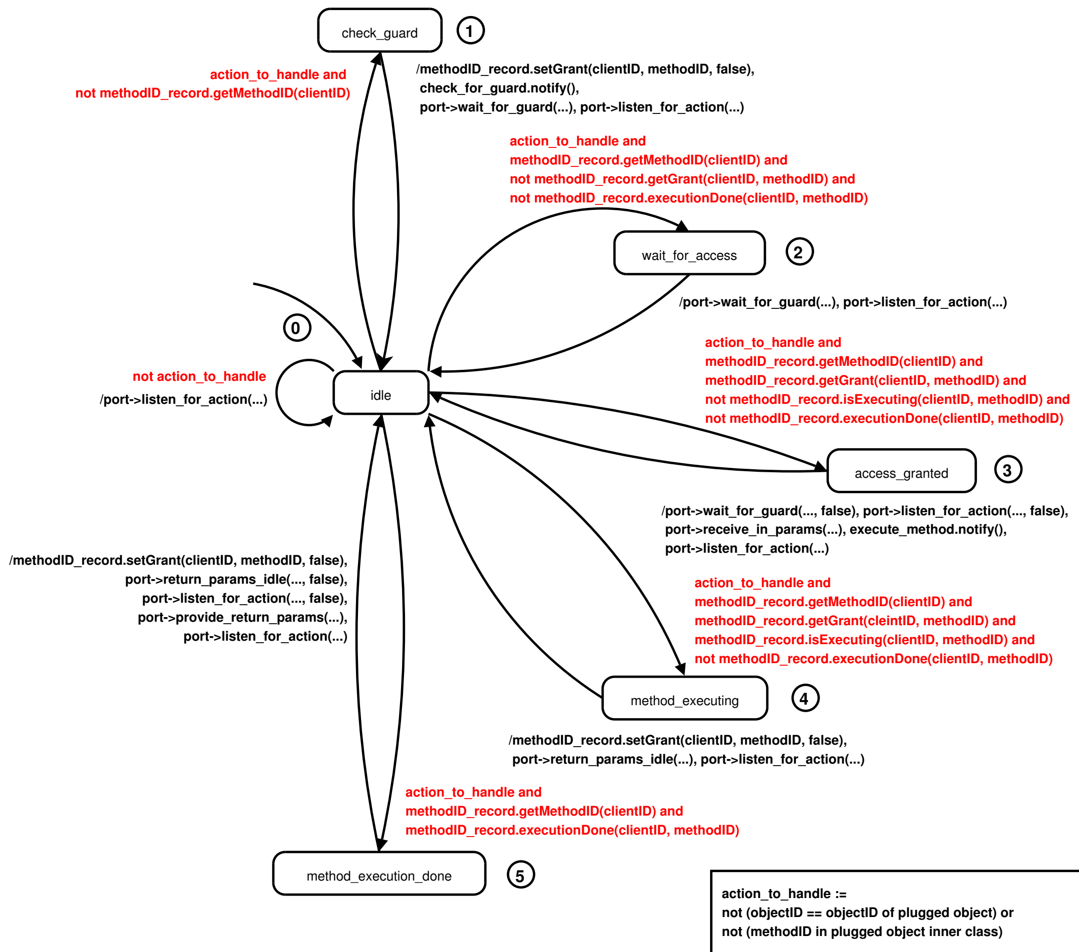


Figure 6.26: `osss_object_socket_port` RMI protocol state machine [44]

The condition *action_to_handle* checks whether the RMI initiated from an `osss_port<osss_rmi_if<IF> >` needs to be handled by the Shared Object inside this `osss_object_socket<...>` or not. The condition *action_to_handle* is an abbreviation for the Boolean condition *not (objectID == objectID of plugged_object)* or for the Boolean condition *not (methodID in Shared Objects inner class)* and checks if the required object is plugged into the Shared Object or if the called method belongs to the plugged object, respectively.

The `listen_for_action(...)` method is called at the end of each Action-List because it reads the current state of the RMI Channel and supplies input for the RMI protocol state machine.

Assuming a method call from an `osss_port<osss_rmi_if<IF> >` that needs to be served by the Shared Object plugged into the considered `osss_object_socket<...>` the state RMI protocol machine is traversed as follows:

1. If the requested method (identified by its `methodID`) has not been registered in the `methodID_record` for the requesting client (identified by its `clientID`) the `check_guard` state is entered. When leaving the `check_guard` state the following actions are performed:

   (a) The `methodID` and the `clientID` of its caller are stored in the `methodID_record`

   (b) The invoke method process is triggered (`check_for_guard.notify()`) and checks if the guard condition for the requested method is true

(c) Call of the wait_for_guard(...) method on the port to the RMI-Channel and thus signalizing the caller that it needs to wait until the guard of the Shared Object evaluates to true.

(d) Call of the listen_for_action(...) method on the port to the RMI-Channel.

2. If the requested method has already been registered for the requesting client and the guard has not been evaluated to true and the execution of this guarded method has not been performed the wait_for_access state is entered. When leaving the wait_for_access state the following actions are performed:

   (a) Call of the wait_for_guard(...) method on the port to the RMI-Channel

   (b) Call of the listen_for_action(...) method on the port of the RMI-Channel

3. If the requested method has already been registered for the requesting client and the guard has been evaluated to true and the requested guarded method is not in execution and the execution of this guarded method has not been finished the access_granted state is entered. When leaving the access_granted state the following actions are performed:

   (a) Call of the wait_for_guard(..., false) method on the port to the RMI-Channel and thus signalizing the caller that the guard has evaluated to true and he can start with the transfer of the method parameters within the next RMI protocol cycle

   (b) Call of the listen_for_action(..., false) method on the port to the RMI-Channel

   (c) Call of the reveive_in_params(...) method on the port to the RMI-Channel and thus copying all method parameters to the *parameter memory* of the osss_object_socket<osss_shared<...> >

   (d) The invoke method process it triggered and executes the requested guarded method whose parameters are available in the param memory (see Figure 6.25)

   (e) Call of the listen_for_action(...) method on the port to the RMI-Channel

4. If the requested method has already been registered for the requesting client and the guard has been evaluated to true and the requested guarded method is in execution and the execution of this guarded method has not been finished the method_executing state is entered. When leaving the method_executing state the following actions are performed:

   (a) For the requesting client and the requested method the grant is set to false inside the methodID_record

   (b) Call of the return_params_idle(...) method on the port to the RMI-Channel and thus signalizing the caller that it has to wait until the return parameter is available

   (c) Call of the listen_for_action(...) method on the port to the RMI-Channel

5. If the requested method has already been registered for the requesting client and the execution of this guarded method has been finished the method_execution_done state is entered. When leaving the method_execution_done state the following actions are performed:

   (a) For the requesting client and the requested method the grant is set to false inside the methodID_record

   (b) Call of the return_params_idle(..., false) method on the port to the RMI-Channel and thus signalizing the caller that the return parameter is available and that it has to read it upon the next RMI protocol cycle

   (c) Call of the listen_for_action(..., false) method on the port to the RMI-Channel

   (d) Call of the provide_return_params(...) method on the port to the RMI-Channel and thus transmitting the return parameter to the caller.

   (e) Call of the listen_for_action(...) method on the port to the RMI-Channel

After having a closer look inside the RMI Layer we will now have a look at the Channel Layer (see Figure 6.21). Most of the internals of the OSSS-Channel will be described in Section 6.5.3. For this reason the next section will focus only on the interfaces of the OSSS-Channel.

The `osss_abstract_basic_channel<...>::readwrite_if`

The `osss_abstract_basic_channel<unsigned int DataWidth, bool BiDirectional>` is the base class or interface for all channels describing a simple point-to-point connection. The first template parameter specifies the data with of the data signal interface inside the channel. The second template parameter specifies whether the point-to-point connection is unidirectional or bidirectional. As described in the previous section channels usable with the RMI protocol need to be at least bidirectional. In the following part of this section we only consider the bidirectional model of the basic channel.

```
1 virtual bool write_blocking(const data_chunk& data) = 0;
2 virtual bool write_blocking(osss_serialisable_object& ser_obj) = 0;
3 virtual bool write_blocking(osss_serialisable_archive& ser_arch) = 0;
4
5 virtual bool read_blocking(data_chunk& data) = 0;
6 virtual bool read_blocking(osss_serialisable_object& ser_obj) = 0;
7 virtual bool read_blocking(osss_serialisable_archive& ser_arch) = 0;
```

Listing 6.20: The `osss_abstract_basic_channel<...>::readwrite_if` interface

The only interface the `osss_abstract_basic_channel<..., true>` provides is the `readwrite_if` shown in Listing 6.20. It provides three different methods for a blocking write and three different methods for a blocking read service. The difference between these read and write methods is the granularity of data they accept. As shown in Figure 6.21 the transfer of method parameters is performed at different granularities dependent on the layer of the OSSS RMI protocol stack.

The `readwrite_if` is implemented by the readwrite transactor inside the channel. The transactor implements the `write_blocking(...)` and `read_blocking(...)` methods by describing how (serialized) objects are transmitted or received using the available low level signal ports.

The `osss_abstract_channel<...>::master_if`

The `osss_abstract_channel<class Arbiter, class AddressDecoder, unsigned int MinDataWidth, unsigned int AddressWidth>` is the base class or interface for all channels describing a multiple master to multiple slave communication (e.g. a shared bus or a crossbar-switch). The first template parameter is used to specify an arbiter class for the channel. Arbitration becomes necessary when multiple masters require access to the same shared medium. The second template parameter can be used to specify a centralized address decoder that is often used in shared buses (e.g. the ARM AMBA bus has a centralized address decoder). The third template parameter specifies the minimal accessible data width (in the most cases this should be eight bit). This information is used to construct the data type of `data_chunk` (more precisely: `typedef std::vector<sc_biguint<MinDataWidth> > data_chunk`). The last template parameter is used to specify the width of the address bus.

For example the Xilinx On-Chip Peripheral Bus used in this thesis implements an `osss_abstract_channel<...>` with the following configuration (compatible to [105]):

- `Arbiter` = `osss_static_priority` or `osss_least_recently_used`,
- `AddressDecoder` = `osss_no_address_decoder`,
- `MinDataWidth` = 8 and
- `AddressWidth` = 32.

```
virtual bool write_blocking(address_type slave_base_addr,
                            const data_chunk& data) = 0;
virtual bool write_blocking(address_type slave_base_addr,
                            osss_serialisable_object& ser_obj) = 0;
virtual bool write_blocking(address_type slave_base_addr,
                            osss_serialisable_archive& ser_arch) = 0;

virtual bool read_blocking(address_type slave_base_addr,
                           data_chunk& data) = 0;
virtual bool read_blocking(address_type slave_base_addr,
                           osss_serialisable_object& ser_obj) = 0;
virtual bool read_blocking(address_type slave_base_addr,
                           osss_serialisable_archive& ser_arch) = 0;
```

Listing 6.21: The `osss_abstract_channel<...>::master_if` interface

The `osss_abstract_channel<...>` provides a master and a slave interface. Listing 6.21 shows the master interface with blocking read and blocking write methods. In addition to the readwrite interface of the `osss_abstract_basic_channel<...>` the methods shown in Listing 6.21 need to address a certain slave since more than a single slave can take part. The granularity of the data has already been explained in the previous sections.

The `osss_abstract_channel<...>::slave_if`

The slave interface of the `osss_abstract_channel<...>` shown in Listing 6.22 is the complement to the master interface. It provides blocking read as well as blocking write methods of the tree different data granularities. Since the slave cannot perform any action on its own, but can only react on a master's request, the `wait_for_action(...)` method is provided.

```
virtual void wait_for_action(const address_type& base_address,
                             const action_type& action) = 0;

virtual bool read_blocking(data_chunk& data) = 0;
virtual bool read_blocking(osss_serialisable_object& ser_obj) = 0;
virtual bool read_blocking(osss_serialisable_archive& ser_arch) = 0;

virtual bool write_blocking(const data_chunk& data) = 0;
virtual bool write_blocking(osss_serialisable_object& ser_obj) = 0;
virtual bool write_blocking(osss_serialisable_archive& ser_arch) = 0;
```

Listing 6.22: The `osss_abstract_channel<...>::slave_if` interface

The `wait_for_action(...)` method returns the address and the kind of action requested by a master. Up to now we only support two action types: `READ_ACTION` and `WRITE_ACTION`. When a `READ_ACTION` is sent by a master the slave has to respond with a call to `write_blocking(...)`. A `WRITE_ACTION` needs to be responded by a `read_blocking(...)` call.

```
osss_port<osss_channel_if<ChannelType::slave_if> > input_port;

osss_memory<ChannelType::data_type, ChannelType::address_type> memory;
ChannelType::address_type address_offset;

void slave_proc() {
  ChannelType::data_chunk data;
  ChannelType::action_type requested_action;
  ChannelType::address_type requested_address;
  wait();

  while(true) {
    input_port->wait_for_action(requested_address, requested_action);

    switch (requested_action) {
      case ChannelType::WRITE_ACTION :
        input_port->read_blocking(data);
        memory.write(requested_address - address_offset, data);
        break;

      case ChannelType::READ_ACTION :
```

```
22          input_port->write_blocking( memory.read(requested_address - address_offset) );
23          break;
24      }
25    }
26  }
```

Listing 6.23: Using an `osss_abstract_channel<...>::slave_if` to implement a simple memory controller

Listing 6.23 shows the usage of an `osss_abstract_channel<...>::slave_if` to implement a simple memory controller. The `slave_proc` process is a `CTHREAD` that calls the `wait_for_action(...)` method on the `input_port` first. The `wait_for_action(...)` method blocks or waits until an action for this memory controller has been detected. After the `wait_for_action(...)` method returned from its call the requested action is evaluated. If a `WRITE_ACTION` has been issued, the slave reads the data from the bus and stores them into the memory at the requested address (modified by a user-defined offset). When a `READ_ACTION` has been issued the slave writes data from the memory back to the channel.

Summary

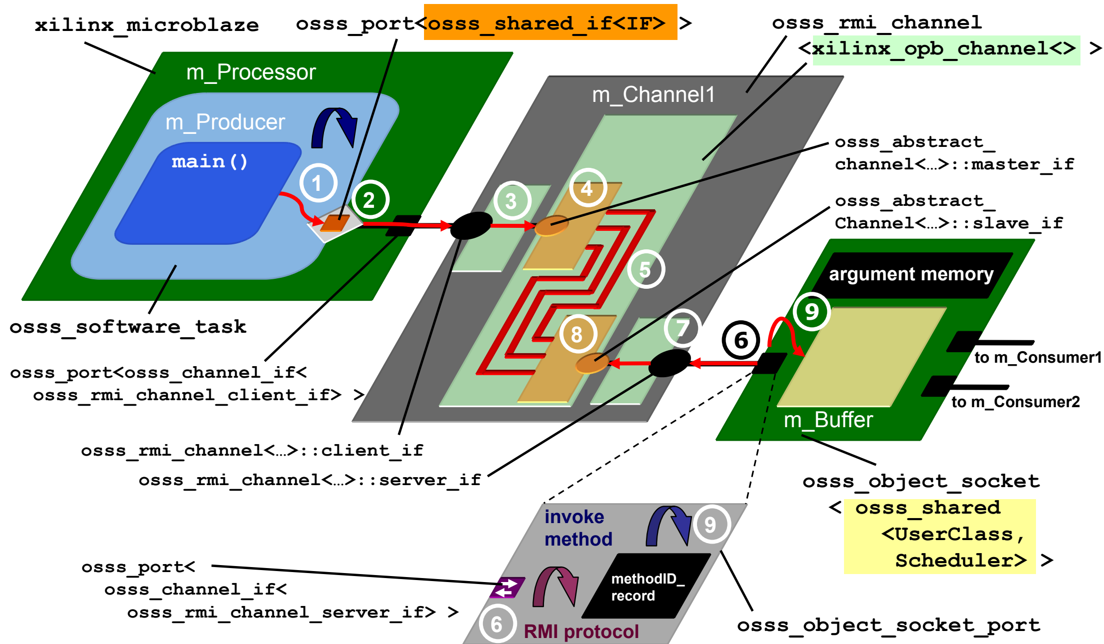


Figure 6.27: Processes communicating through OSSS RMI [44]

Figure 6.27 depicts the RMI protocol stack of the HW/SW interface from Figure 6.20. The producer is mapped onto a Xilinx MicroBlaze processor that communicates with the buffer Shared Object by using the RMI protocol over a Xilinx OPB Channel.

When a remote method is called on a local port `osss_port<osss_shared_if<IF> >` ①, the RMI protocol becomes initiated. This implies a sequence of the following operations:

- ② serialization of the arguments of the called method (at this point the `serialise()` method is called on each argument of a user-defined class; for all built-in types a predefined serialization action takes place),
- ③ the client transactor of the OSSS-RMI-Channel performs the RMI protocol which includes the submission of the caller's client ID, the callee's object ID together with the ID of the called method and the serialized arguments,

- ④ the master transactor of the Xilinx OPB Channel establishes the signal level communication and transforms the method based communication from the RMI-Channel client transactor into a signal based ⑤ OPB protocol compliant transaction.

The RMI server process inside the object socket port ⑥ performs a method call on the RMI-Channel server interface ⑦ parallel to steps ① - ④. This call is translated by the OPB slave transactor ⑧ to a "listening for action" on the shared bus signals ⑤.

When a remote method call to the "listening" object socket is detected the client ID, method ID and the serialized arguments are received by the protocol process ⑥. The serialized method arguments are written to an argument memory inside the object socket. In the last step the `deserialise()` method (as counterpart to `serialise()`) is called on all arguments and the method call on the buffer Shared Object is performed ⑨.

The same protocol has to be performed on the way back from the remote object to the caller, when a method with a non-void return argument has been called. Moreover, this protocol would have been the same for a hardware module instead of a software task as initiator of the communication.

6.5.3 OSSS-Channels[10]

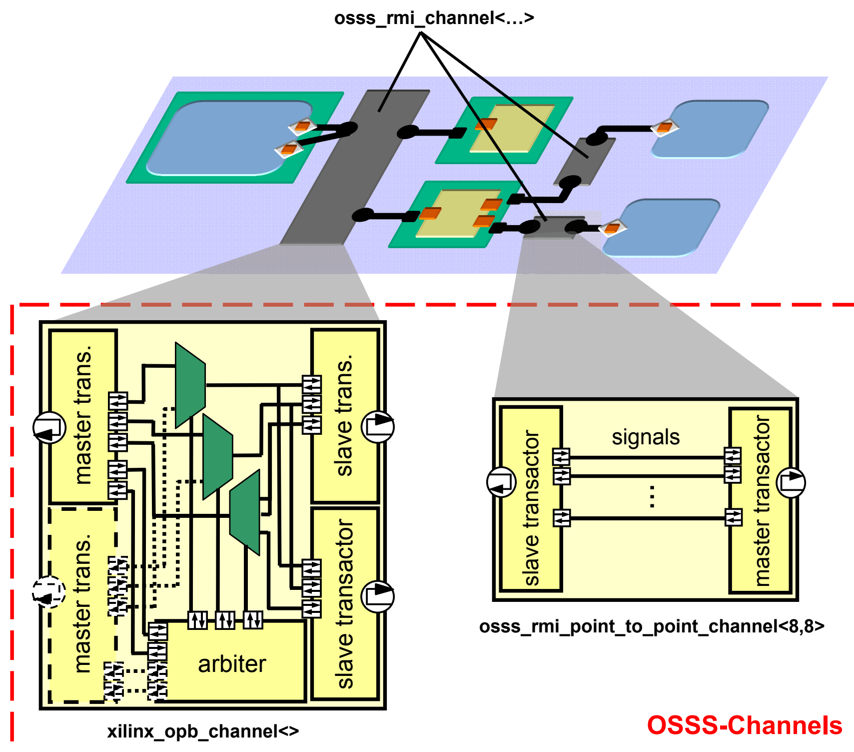


Figure 6.28: Relationship between OSSS-Channels and RMI containers

Figure 6.28 demonstrates the relationship between OSSS-Channels and the generic RMI container. OSSS-Channels define the internal structure of the RMI channel. Basically an OSSS-Channel consists of transactors which translate from a method-based communication to a communication on signal level. The method-based communication follows the generic RMI protocol. The OSSS-Channel master transactor translates a sequence of read or write transactions to a sequence of signal value changes. These signal changes follow a certain communication protocol (e.g. an IBM OPB protocol). The OSSS-Channel slave transactors serve the same purpose for the way back from a certain physical communication protocol to the generic RMI protocol. The signals between the master and slave transactors can be wired in

[10]This section is based on own previous work [77, 78].

a flexible way to enable different communication topologies, ranging from point-to-point, over shared bus to a cross-bar switch topology.

The OSSS-Channel approach is a concept that enables the designer to model the link and physical protocol layers independently from the RMI layer (representing the network, transport, session and presentation layers). Building the OSSS-Channel on top of the object oriented features provided by OSSS enables the transfer of both simple data types (excluding pointers (*) and references (&)) and objects (containing data members of simple and complex type, including class types and array types).

In the following subsections the OSSS-Channel approach is introduced. It starts with a general overview by presenting the key concepts, followed by a modeling example of a simple point-to-point connection using an OSSS-Channel. After describing these basics, multi-master OSSS-Channels are introduced. This is done by the example of the `xilinx_opb_channel<...>` that has already been used in the previous sections.

6.5.3.1 Key Concepts

The OSSS-Channel concept is based on the separation of port, interface and implementation known from SpecC and SystemC channels and encapsulates a user-defined protocol and physical connection. The separation principle allows the designer to choose from different implementations of a channel as long as they implement the same interfaces. This concept allows the separation of the application from communication details. Additionally, it eases the exploration of different communication protocols for a certain design through replacing different channels of the same interfaces ("plug and play"). This is assisted by storing OSSS-Channels in the library of the *Virtual Target Architecture* for easy reuse.

The OSSS-Channel provides mechanisms to generate the necessary communication infrastructure. The internal structure consists of transactors, arbiters, address decoders and a signal level interconnect network. The generation of the internal structure is invoked by the binding of an OSSS-Port of a module to an OSSS-Channel or by the binding to an RMI-Channel container that encapsulates an OSSS-Channel. The transactors provide a method based interface to the outside of the channel and a signal based interface to the signal network inside the channel. Transactors implement the method interface and utilize the signal interface to realize the physical layer protocol. If more than one master is connected to the channel an arbiter handles the requests. The arbitration mechanism is specified by a user-defined scheduling policy (similar to the scheduling of Shared Objects, actually the same scheduling algorithms can be used).

The main features of the OSSS-Channel are:

- allows to write user-defined link and physical layer protocols
- allows to connect multiple master and slave modules to the same instance of an OSSS-Channel
- offers a mechanism to model communication between different modules through method calls
- allows to transfer any valid C++/SystemC data types including classes (no pointers (*) and references(&))
- follows the hierarchical channel design concept known from SpecC and SystemC (separation of port, interface and implementation)
- supports the automatic (and dynamic[11]) generation/construction of the communication network inside of an OSSS-Channel (invoked by the binding of an OSSS-Port to an OSSS-Channel). This includes the generation of:
  - transactor instances
  - arbiter instances with user-defined schedulers
  - address decoders

[11] during SystemC elaboration phase

  – signal network connecting these channel internals

- allows to create/expand a library of different channel implementations
- can be directly translated into a synthesizable RTL representation

6.5.3.2 A Simple Point-To-Point Channel

As an introductive example, we will present an OSSS-Channel implementing a unidirectional point-to-point connection as shown in Figure 6.29. The example consists of two modules (producer and consumer) which communicate directly via an `osss_simple_point_to_point_channel<...>`. The producer initiates write transfers on the channel while the consumer reads from the channel and consumes the transferred data.

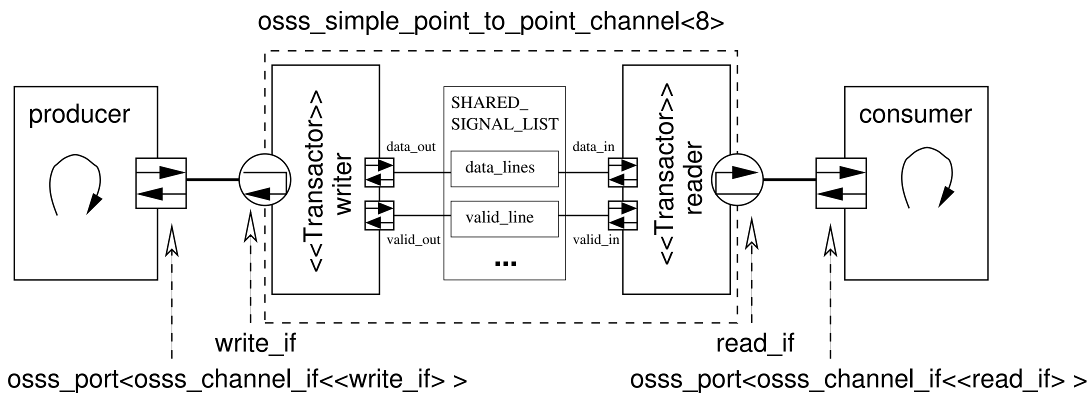


Figure 6.29: OSSS-Channel model of a unidirectional point-to-point connection [78]

Figure 6.30 shows the layer concept of the OSSS-Channel used to implement the `osss_simple_point_to_point_channel<...>`. The presented point-to-point channel implements the `osss_abstract_basic_channel` that contains two inner interface classes, `write_if` and `read_if`. These interfaces define the methods that have to be implemented by the transactors of the channel implementation.

In Figure 6.30 these classes are located at the *Channel Interface Layer*. The design methodology of the OSSS-Channel requires each user-defined channel to implement a certain pre-defined method interface. Otherwise the separation of application and communication and the exchangeability of channels, providing the same interface but containing different protocol and signal level implementations, cannot be guaranteed.

The *Channel Implementation Layer* contains the transactors that transform the method based communication initiated from outside the channel to a signal based communication inside the channel.

The bottom layer, called *Channel Access Layer*, contains the `osss_port_to_channel<...>` port that serves two purposes. Firstly, it creates the channel internals (transactors and their connections to the inner channel signal interface) upon port to interface binding. And secondly, it provides the `operator->()` that is used for calling the methods implemented by the transactors. In the SystemC terminology this kind of module to channel communication is called *Interface Method Call* (IMC).

The uppermost layer is called *Channel Generation/Service Layer*. It provides services for the signal level connection between transactors inside the channel and is described in more detail later in this section.

In our simple example in Figure 6.29 the ports of the producer and consumer modules are bound to the point-to-point channel. Each port is declared using the `osss_port<osss_channel_if<...> >` whose template parameter specifies the method interface for accessing the channel (see Listing 6.24). The implementation of the method interface classes is done inside the channels' transactors. Except for the channel internals (transactors and their signal based interconnection), our port to interface binding is compliant to the SystemC port to interface binding methodology. Thus, the point-to-point channel can be used like an

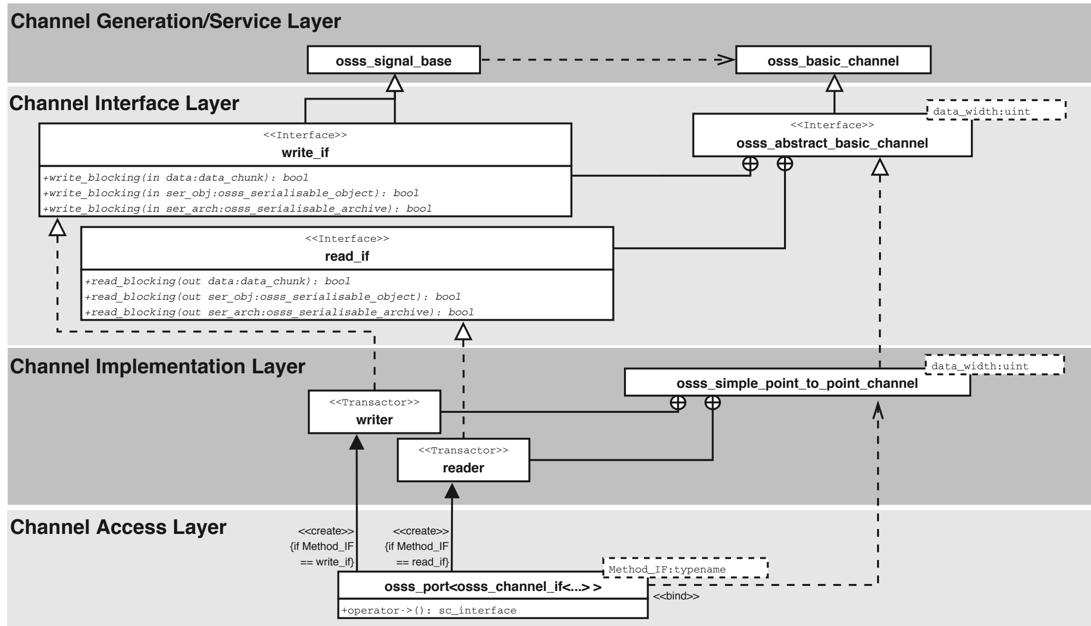


Figure 6.30: Layer concept of the OSSS-Channel methodology used to implement a simple point-to-point channel [78]

`sc_signal` with the possibility to transfer any user-defined data type and defining the physical implementation via a constrained set of signals[12].

```
1  typedef osss_simple_point_to_point_channel<8> Channel_t;
2
3  OSSS_REGISTER_TRANSACTOR(Channel_t::writer, Channel_t::write_if);
4  OSSS_REGISTER_TRANSACTOR(Channel_t::reader, Channel_t::read_if);
5
6  SC_MODULE(producer) {
7    osss_port<osss_channel_if<Channel_t::write_if> > out;
8
9    void producer_proc() {
10     ...
11     out->write_blocking(...);
12     ...
13   }
14   ...
15 };
16
17 SC_MODULE(consumer) {
18   osss_port<osss_channel_if<Channel_t::read_if> > in;
19
20   void consumer_proc() {
21     ...
22     in->read_blocking(...);
23     ...
24   }
25   ...
26 };
```

Listing 6.24: Point-to-point example *Channel Access Layer*

As already shown in Figure 6.30 the `write_if` requires the implementation of the `write_blocking(...)`[13] method. This blocking interface method takes an object of type `osss_serialisable_object`. That means each object which should be transferred via an

[12] An `sc_signal` of a user-defined data type has the bit with of the user-defined data type (i.e. the bit width resulting from a concatenation of all data members).

[13] Blocking is meant in the sense of calling a method and not returning from that call until its execution has been finished.

OSSS-Channel has to be derived from the `osss_serialisable_object` class. Thus, each class inheriting from `osss_serialisable_object` has to implement the purely virtual methods `serialise()` and `deserialise()`. These methods declare which members of the class have to be serialized and de-serialized (see Section 6.6).

The `osss_serialisable_object` class enables the serialization and de-serialization of a data object in order to transfer it by the low level protocol via an arbitrary bit width constrained set of `data_line` signals inside the channel. The width of the `data_lines` signal has to be determined during declaration of the `osss_simple_point_to_point_channel<...>` through its template parameter[14] (see Listing 6.24). This use of the serialisable object base class is necessary to allow the transfer of arbitrary (user-defined) data types by every OSSS-Channel without a modification of the protocol inside of the channel. Thus, guaranteeing the separation concept of application/computation and communication.

```
1  class writer : public base_type::write_if {
2  public:
3    sc_out< sc_uint< dataWidth > > data_out;
4    sc_out<bool> valid_out;
5
6    OSSS_GENERATE {
7      osss_connect(osss_reg_port(data_out),
8                   osss_shared_signal("data_lines"));
9
10     osss_connect(osss_reg_port(valid_out),
11                  osss_shared_signal("valid_line"));
12   }
13
14   virtual bool write_blocking(osss_serialisable_object& ser_obj) {
15     valid_out = true;
16     wait(2);
17     valid_out = false;
18     ser_obj.serialise_obj();
19     while (!ser_obj.complete()) {
20       ser_obj.write_chunk_to_port(data_out);
21       wait();
22     }
23     return true;
24   }
25 };
26
27 class reader : public base_type::read_if {
28 public:
29   sc_in< sc_uint< dataWidth > > data_in;
30   sc_in<bool> valid_in;
31
32   OSSS_GENERATE {
33     osss_connect(osss_shared_signal("data_lines"),
34                  osss_reg_port(data_in));
35
36     osss_connect(osss_shared_signal("valid_line"),
37                  osss_reg_port(valid_in));
38   }
39
40   virtual bool read_blocking(osss_serialisable_object& ser_obj) {
41     while(valid_in.read() == true) wait();
42     while(valid_in.read() == false) wait();
43     while (!ser_obj.complete()) {
44       ser_obj.read_chunk_from_port(data_in);
45       wait();
46     }
47     ser_obj.deserialise_obj();
48     return true;
49   }
50 };
```

Listing 6.25: Point-to-point example *Channel Implementation Layer*

[14]Thus `osss_simple_point_to_point_channel<8>` declares a point-to-point channel with a `data_lines` signal width of 8 bits.

We will now have a closer look at the channel internals, i.e. the implementation of the transactors and their signal level interconnection.

Listing 6.25 shows the implementation of the `writer` and the `reader` transactor of the *Channel Implementation Layer*. The transactors themselves describe the communication protocol by implementing the interfaces of the *Channel Interface Layer*. The `writer` transactor implements the `write_if` and the `reader` transactor implements the `read_if`. Figure 6.30 shows that both, the write and the read interface inherit from the `osss_signal_base` class of the *Channel Generation/Service Layer*. This base class provides services for the signal based interconnection of the transactors.

Each transactor has an RTL SystemC signal interface consisting out of ports and information about their binding to the signal network inside of the channel. This RTL signal interface is necessary to provide the synthesizable low level port interface which is used to establish the signal level communication between the modules connected to the channel. The signal level communication protocol itself is encapsulated by the transactors. They implement the `write_blocking(...)` and the `read_blocking(...)` methods by describing how (serialized) objects are transmitted or received using the available signal ports.

In our example a user-defined protocol with two signal level ports (`data_out/in` and `valid_out/in`) for each transactor is chosen. The `valid` port is used to manage the control and the `data` port is used to transfer the serialized object. The width of the data port is defined by a channel template parameter. These ports are bound to the channel's communication network inside the `OSSS_GENERATE` block statement.

In this section the ports of each transactor are connected by the `osss_connect(...)` service. This connection service is used inside the `writer` transactor to connect the `data_out` and the `valid_out` ports to a shared signal called `data_lines` and `valid_line` respectively. It is used the other way round in the `reader` transactor, where connections are defined from the shared signals to the respective ports. In Figure 6.29 this signal based interconnection is visualized by the `SHARED_SIGNAL_LIST` which contains the shared signals `data_lines` and `valid_line`.

The `write_blocking(...)` method of the `reader` transactor generates a control signal that marks the beginning of a data transfer and transmits it via the `valid_out` port. After this notification the serializable object's attributes are written to a bit vector by invoking the `serialise_obj()` method. This bit vector is divided into chunks of the size of the `data_out` port. The `write_chunk_to_port(...)` method is executed in a loop until the transmission of the whole bit vector has been completed.

The `read_blocking(...)` method of the `reader` transactor is the inverse of the `write_bocking(...)` method described above. It awaits the beginning of a transmission and collects the received data chunks until the serializable object is received completely. In the last stage it rebuilds the serializable object by assigning the bit vector back to the attributes of the respective object (`deserialise_obj()`).

The `writer` and `reader` transactors inside of the channel are automatically instantiated and connected to the signal network (shared signals) when a port of the type `osss_port<osss_channel_if<...> >` is bound to the channel. The `OSSS_REGISTER_TRANSACTOR` macro is used to define which transactor has to be generated in dependence of the method interface it implements (see Listing 6.24).

When transforming an OSSS-Channel into a synthesizable RTL representation the transactors become part of the connected modules. In the modeling phase a transactor belongs to the OSSS-Channel while after high level synthesis it belongs to the connected module. Figure 6.31 shows the synthesis result of the unidirectional point-to-point channel presented during this section. Both `writer` and `reader` transactors are "inlined" into the producer and consumer module (denoted by p* and c*). Or more precisely, the protocol state machines of the transactors become inlined into the producer and consumer state machines. The ports of the transactors become ports of the producer and consumer modules. The shared signal list from inside the OSSS-Channel becomes converted into a simple signal connection.

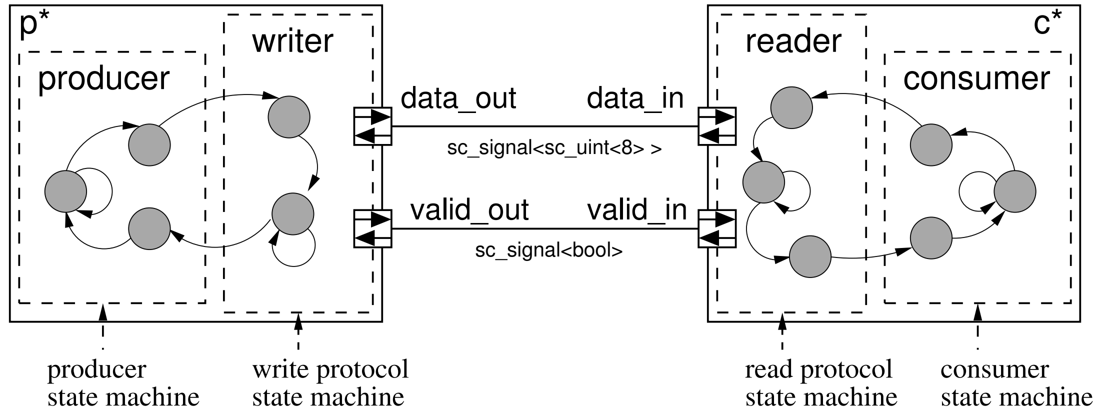


Figure 6.31: Synthesis result of the unidirectional point-to-point channel (`osss_simple_point_to_point_channel<8>`) [78]

6.5.3.3 A Channel with Arbitration

In the previous section an OSSS-Channel constituting a simple point-to-point connection was introduced. Now we consider an OSSS-Channel where multiple master modules use the same transactor type to initiate transactions. Generally, if several processes want to initiate transactions on the same channel, an arbitration mechanism becomes necessary. This section introduces an OSSS-Channel with a generic arbitration mechanism.

The usage of OSSS-Channels with arbitration is very similar to simple point-to-point connection channels. It uses the same binding, generation and communication mechanisms as discussed previously. Additionally, the user has to distinguish between master and slave method interfaces which provide more generation and connection services for the signal connection inside the channel. This rich set of connection services enables the channel designer to describe different communication architectures like buses or crossbar switches. Besides these services it is possible to integrate user-defined arbiters and user-defined address decoders typically found in state-of-the-art bus implementations [145, 84].

A multi master/slave OSSS-Channel which is used by several modules can implement different communication protocol schemes defined in its transactors. Each module initiating transactions (master module) on such an OSSS-Channel uses a master transactor. Modules that only react on initiated transactions (slave module) use a slave transactor.

There are two possible approaches to activate a slave module upon a master request. One relies on a central address decoder activating the passively waiting slaves. The other is to keep the slaves listening to the bus and let them activate themselves when requested (broadcast). We will call the first address decoding centralized address decoding while the second one will be considered as distributed address decoding.

The OSSS-Channel supports both centralized and distributed address decoding. Either, by explicitly specifying an address decoder or by using the predefined `osss_no_address_decoder` dummy, which indicates the use of a distributed address decoding scheme.

Figure 6.32 shows the software architecture concept used to implement a channel with arbitration. Comparing it to Figure 6.30, it contains additional classes that refine the *Channel Generation/Service Layer*. This refinement is necessary because of the introduced master-/slave communication scheme, the more complex inner channel signal interconnections and the arbiter and address decoder integration. The *Channel Interface Layer* serves the same purpose as in the point-to-point channel. It primary enables the separation of application and communication and secondary the exchangeability of the *Channel Implementation Layer*. The `master_if` declares blocking write and read methods in the same manner as the write and the read interfaces from Figure 6.30. The additional `slave_base_address` parameter is used to specify the destination of the data transfer. Since a slave does not invoke any action on its own, the `slave_if` contains the additional `wait_for_action(...)` method. Called once it waits until the corresponding slave is accessed by a master and returns the base address and the required action (could either be a read- or a write-action) requested by the master. Dependent on the action, the corresponding

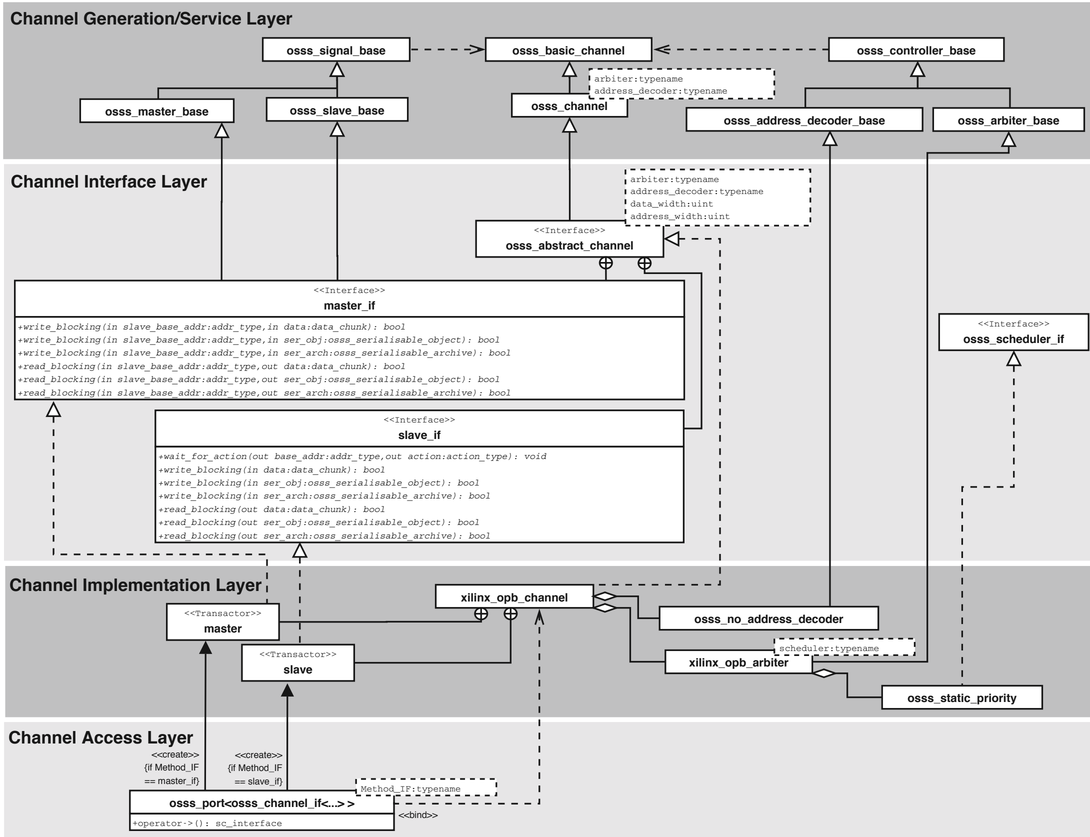


Figure 6.32: Layer concept of the OSSS-Channel methodology used to implement a channel with arbitration [78]

`write_blocking(...)` or `read_blocking(...)` method of the slave is executed.

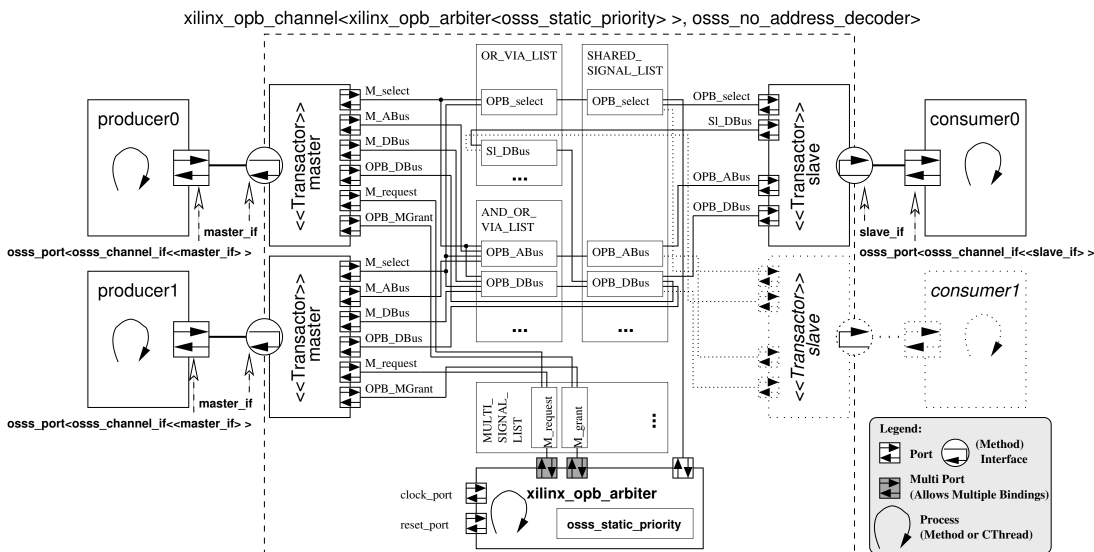


Figure 6.33: The OSSS-Channel model of a channel with arbitration (`xilinx_opb_channel<...>`) [78]

Figure 6.33 depicts an extract of the OSSS-Channel simulation model internals using a bus architecture with distributed address decoding as specified in [151, 105]. It shows three modules connected to different transactors. The `producer0` and `producer1` modules are connected

to a master transactor and communicate with it through methods specified in `master_if`. Module `consumer0` is a slave module and is connected to a slave transactor. The low level signal interfaces of the master and the slave transactor are significantly different from each other. Master transactors are not only connected to the shared signals for data and address communication but also to an arbiter component. In addition to the `SHARED_SIGNAL_LIST`, used to connect the `writer` and the `reader` transactor in Figure 6.29, the `OR_VIA_LIST`, `AND_OR_VIA_LIST` and `MULTI_SIGNAL_LIST` have been introduced in Figure 6.33. The `OR_VIA_LIST` contains elements that perform a logical "OR" ($\vee$) on its input signals. The `AND_OR_VIA_LIST` contains elements that perform a logical "AND" ($\wedge$) on a certain pair of input signals and afterwards a logical "OR" ($\vee$) on all the outputs of the previously AND-connected stage. The `MULTI_SIGNAL_LIST` contains elements that are collecting signals which are not shared but unique to each transactor (e.g. the request and grant signals from the master transactors to the arbitration unit). All these lists and their components are part of the *Channel Generation/Service Layer*. The lists and the components are scalable which gives the flexibility to add an arbitrary number of master or slave transactors to the channel (see `consumer1` and its slave transactor for an illustration of what happens if a second slave is attached to the channel).

Another interconnection service provided by the *Channel Generation/Service Layer* is the `osss_mux_via` that models a scalable multiplexer. The user has to specify at least a single output, a single control and one or arbitrary input signals. While the On-Chip Peripheral Bus (OPB) modeled in Figure 6.33 uses the `and_or_via` to implement a distributed multiplexer, the `osss_mux_via` is intended to model a centralized multiplexer.

The `xilinx_opb_arbiter` is a modules with a process to implement a user-definable scheduling policy. To retain the interchangeability of schedulers, each scheduler that should be used inside such an arbiter block has to implement the `osss_scheduler_if`. This scheduler interface belongs to the *Channel Interface Layer* (see Figure 6.32) and is the same scheduler interface as used for Shared Objects.

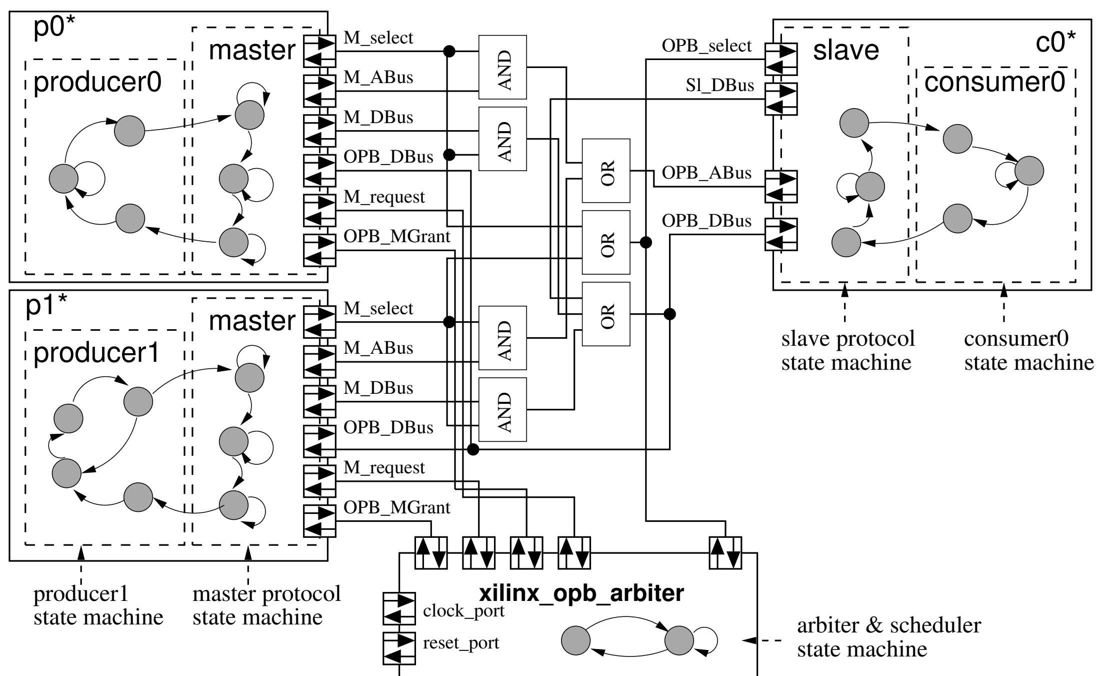


Figure 6.34: Synthesis result of the channel with arbitration [78]

Figure 6.34 shows the synthesis result for the OSSS-Channel model of the multi master/slave design. Each transactor once contained in the `xilinx_opb_channel<...>` is now part of the respective module (denoted by `p0`*, `p1`* and `c0`*). The interconnections inside the channel have been transformed into a communication network consisting of signals, and-gates and or-gates.

Assuming that both, the module and the associated transactor, are described in OSSS they form a synthesizable module or entity which is directly connected to the generated interconnection

network. Like in the synthesis result of the simple point-to-point connection, the master and slave protocol state machines are directly connected to the respective producer/consumer state machines (referred to as method inlining during the synthesis step).

The `xilinx_opb_arbiter` together with its embedded `osss_static_priority` scheduler are synthesized to a single state machine. The inlining of the scheduler to the arbiter state machine works in almost the same manner as the inlining of the producer/consumer with the master/slave state machines.

The multi ports of the arbiter used by the `request` and `grant` signals of the master transactors have been transformed to single ports. The mandatory `clock` and `reset` ports of the producer and consumer module have been omitted in Figure 6.34[15].

6.6 Mapping[16]

The mapping step from the *Application Layer* to the *Virtual Target Architecture Layer* replaces hardware components by OSSS modules, maps software tasks onto processors and wraps Shared Objects by so called Object Sockets. Communication links between software tasks or hardware modules and Shared Objects are replaced by RMI-Channels.

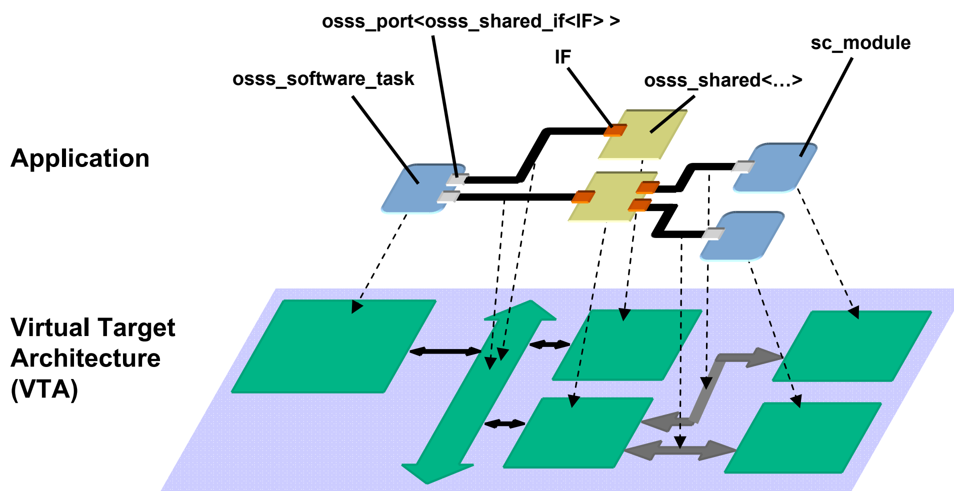


Figure 6.35: OSSS Application to Virtual Target Architecture Mapping (top-down) [44]

The mapping of Hardware Modules, Software Tasks, and communication links from the Application Layer Model to an Execution Platform can be performed under different constraints:

top-down: In a top-down approach the components and the architecture of the target platform is not fixed. Architecture component allocation and binding of application layer hardware modules, software tasks, and communication links to these components is an iterative process to find the optimal execution platform. This approach requires a flexible target architecture model to express different component allocations and interconnection topologies among these components.

bottom-up: In the bottom-up approach the target architecture is more or less fixed. Take for example a System on Chip consisting out of general purpose software processor part including standard peripherals. In this case the application model needs to be highly flexible to be mapped onto the pre-defined architecture.

However, enabling both approaches, a flexible and target architecture independent Application Layer model and a configurable and extendable Architecture Layer model is needed. Both approaches have in common that the mapping of abstract communication links from Application

[15] `clock` ports of a module connected to a channel must be in the same clock domain (i.e. driven by the same physical clock)

[16] This section is based on own previous work [44].

Layer models to physical communication channels in Virtual Target Architecture models is necessary. In the top-down approach this step is usually called communication refinement. An abstract communication channel is refined to a concrete protocol implementation on a physical channel. For the bottom-up approach the term "communication mapping" seems to be more appropriate since the channel implementation is not derived directly from the application requirements.

The OSSS methodology is targeted on the mapping of an Application to a Virtual Target Architecture without[17] the need for a manual communication refinement. Starting with an application model consisting of communicating processes using method calls, several manual refinement steps are needed to end up with a synthesizable[18] description of the application. These steps involve:

- hardware/software partitioning (which parts of the application or algorithm are implemented in software, and which parts are implemented in hardware),
- system architecture definition (allocation of architectural resources like processor types, their number, allocatable hardware resources and memories that can be used for the mapping of the application [top-down], or definition of a fixed execution platform [bottom-up]),
- refinement of the hardware and software module behaviour (when starting from an algorithmic description the refinement to dedicated hardware is called behavioural synthesis. It usually performs scheduling, allocation and binding to a constrained set of hardware resources),
- and communication refinement (transferring the abstract (e.g. transaction or method based) communication to a signal based communication channel implementing a defined protocol).

With OSSS we enable hardware/software partitioning decisions by supporting the designer in finding the "best" solution by manually exploring different application partitionings. The OSSS Application Layer modeling elements (see Section 6.4) can be plugged together easily using Shared Objects as abstract communication medium.

Furthermore, we do not consider behavioural synthesis which converts algorithmic descriptions into RTL structures. Here we suppose that the behavioural synthesis step has either been performed manually or by an external tool. The refinement of the software part is another issue we do not address directly with OSSS. A software task which has been developed and tested on a host system, different from the specific target processor, can show a different behaviour after cross-compilation and execution on the embedded target processor. To avoid these portability problems we advise the designer to make use of the portable subset of the C++ language [207]. Non-portable language constructs should be avoided as much as possible.

The main issue we address with the OSSS methodology is communication refinement. Starting from a universal method based hardware/software and hardware/hardware communication we end up with a synthesizable signal based implementation.

In the following sections the Application Layer to Virtual Target Architecture Layer model mapping is described in more details using the simple producer/consumer example, introduced earlier. After the mapping description an example using different platform configurations an mappings for architecture exploration is presented.

6.6.1 Mapping the Consumer/Producer Design Example

In the following section we will map the simple consumer/producer example that has already been presented in Section 6.4 to the *Virtual Target Architecture Layer*. The upper part of Figure 6.36 shows the *Application Layer* consisting of a producer software task, a Shared Object that contains a user-defined `FIFO` class (in this example is parametrized to hold up to 10 objects of type `Packet`) and a consumer hardware module. The lower part of Figure 6.36 shows the

[17] or at least with minimal manual effort

[18] In our definition "synthesizable" means, each application component can be further processed by state-of-the-art tools like cross-compilers (for the software part) or RTL synthesis tools (for the hardware part)

Virtual Target Architecture Layer, where the mapping and the communication refinement is accomplished.

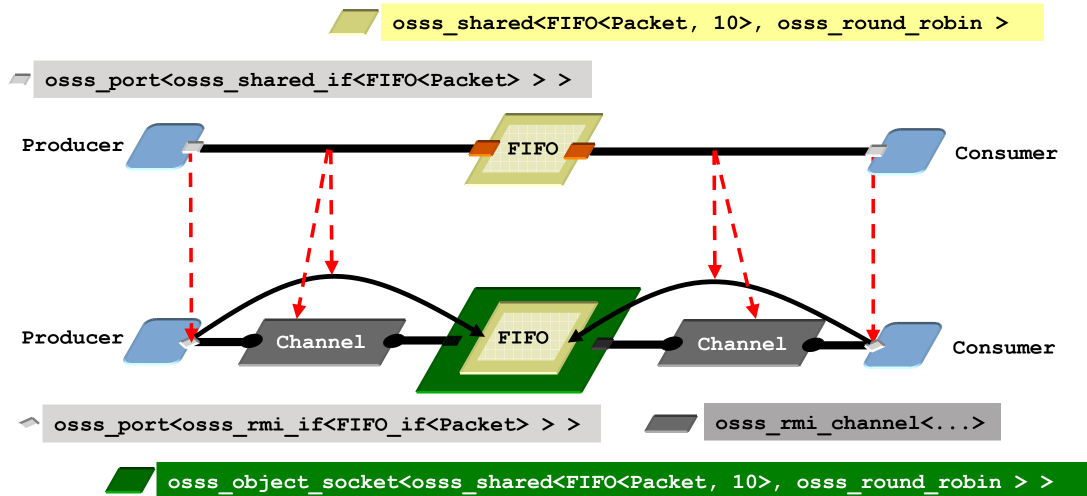


Figure 6.36: Communication refinement of the producer/consumer example [44]

In the following we will perform a stepwise communication refinement of the producer/consumer example. To gain a better understanding about the amount of effort a designer has to spend, some code snippets will be presented. Whenever code from the *Application Layer* needs to be changed or extended we highlight these parts in red color.

As stated above, we assume that the hardware/software partitioning has already been performed (producer implemented as software, consumer implemented as hardware). Additionally, we assume that the behaviours of the producer, the FIFO, and the consumer are already specified in a synthesizable way. With these prerequisites the following refinement steps (the order is negligible) have to be performed:

1. Change all custom hardware modules of type `sc_module` to type `osss_module`.
2. Change the OSSS Port interfaces of all Software Tasks and Hardware Modules from `osss_shared_if<`*IF*`>` to `osss_rmi_if<`*IF*`>`.
3. Build or generate `osss_rmi_if<...>` stubs for each Shared Object interface (for the producer/consumer example these are the `FIFO_put_if<...>` and `FIFO_get_if<...>`).
4. Equip all user-defined data types with serialization support (in the producer/consumer example it is the `Packet` class).
5. Define a custom OSSS-Channel or use a pre-existing OSSS-Channel from the Architecture Class Library (in the consumer/producer example the `xilinx_opb_channel` and the `osss_rmi_point_to_point_channel` are used).
6. Assemble the top-level design.

6.6.1.1 The `osss_rmi_if<...>` interface stub

All ports of kind `osss_port<osss_shared_if<...> >` need to be replaced by ports capable of performing RMI (i.e. `osss_port<osss_rmi_if<...> >`). Listing 6.26 and Listing 6.27 highlight the replaced ports (changes regarding the *Application Layer* are marked in red color).

```
1 OSSS_SOFTWARE_TASK(Producer) {
2   osss_port<osss_rmi_if<FIFO_put_if<Packet> > > output;
3
4   OSSS_SW_CTOR(Producer) { }
5
6   virtual void main() { ... }
```

```
7 };
```

Listing 6.26: Producer with RMI port

```
1  OSSS_MODULE(Consumer) {
2    sc_in<bool> pi_bClk;
3    sc_in<bool> pi_bReset;
4
5    osss_port<osss_rmi_if<FIFO_get_if<Packet> > > input;
6
7    sc_out<Packet>  po_Packet;
8
9    SC_CTOR(Consumer) {
10     SC_CTHREAD(main, pi_bClk.pos());
11     reset_signal_is(pi_bReset, true);
12   }
13
14   void main();
15 };
```

Listing 6.27: Consumer with RMI port

Besides these port modifications, the designer has to provide an implementation of the `osss_rmi_if<...>` stub for each Shared Object interface. Concerning the other code of the producer and the consumer nothing has to be changed because we assume their behaviour is already synthesizable. Listing 6.28 shows the implementation of the RMI stub for the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interfaces.

The stubs are derived from the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interface classes. This is necessary because we need to provide a dedicated stub for each method specified in the interface classes. The stub is created by the `OSSS_OBJECT_STUB_CTOR( _IF_type_ )` constructor, where `_IF_type_` is the type of the interface that needs to be implemented by this stub (It is usually the interface class the stub is derived from. In our example this are the `FIFO_put_if<ItemType>` and `FIFO_put_if<ItemType>` interfaces).

Hint: When using complex types inside macros it is good practice to make a `typedef` before (see lines 4 & 5 in Listing Listing 6.28). Consider the following example: Given a macro that takes a single argument like `#define MY_MACRO( _type_ )`, and type `My_Template_Type< a, b, c >`; using the macro `MY_MACRO( My_Template_Type< a, b, c > )` leads to a compilation error, since the pre-processor treats each comma as a separated argument. Using `typedef My_Template_Type< a, b, c > my_template_t` and instead `MY_MACRO( my_template_t )` leads to the desired behaviour.

Each method specified in the interface classes needs to be declared by either the `OSSS_METHOD_VOID_STUB(...)` or by the `OSSS_METHOD_STUB(...)` macro. The first macro is used for methods with a void return type while the second macro is used for methods with a non-void return type. These macros are very similar to the `OSSS_GUARDED_METHOD_VOID(...)` and the `OSSS_GUARDED_METHOD(...)` macros used inside the Shared Object's user-defined class implementation. The main difference is that the stub macros do not have a guard condition parameter.

```
1  template<class ItemType>
2  class osss_rmi_if<FIFO_put_if<ItemType> > : public FIFO_put_if<ItemType> {
3    public:
4      typedef FIFO_put_if<ItemType> base_type;
5      OSSS_OBJECT_STUB_CTOR(base_type);
6
7      OSSS_METHOD_VOID_STUB(put, OSSS_PARAMS(1, ItemType, item));
8      OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
9      OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
10 };
11
12 template<class ItemType>
13 class osss_rmi_if<FIFO_get_if<ItemType> > : public FIFO_get_if<ItemType> {
14   public:
15     typedef FIFO_get_if<ItemType> base_type;
16     OSSS_OBJECT_STUB_CTOR(base_type);
17
```

```
18     OSSS_METHOD_STUB(ItemType, get, OSSS_PARAMS(0));
19     OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
20     OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
21 };
```

Listing 6.28: RMI stubs for the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interfaces

The `osss_rmi_if<...>` acts as a stub or proxy to the remote object. Each method stub macro generates the appropriate code that is needed to perform a remote method invocation. This includes the determination of the method ID for the called method, the serialization of all parameters, the transmission of this data through the bound RMI-Channel and the de-serialization of the return parameter (for non-void methods only) received from the RMI-Channel.

6.6.1.2 Serialisation of user-defined data types

All data types that should be transferred via RMI have to be serializable. That means each data type or user-defined class needs to be decomposable into chunks of a specific size in order to be transmittable through any channel of arbitrary data width. The OSSS library has support for all built-in C and C++ data types. Moreover, it supports all synthesizable SystemC data types. When dealing with user-defined data types like structs or classes some manual effort is required to make them serializable.

Listing 6.29 shows the user-defined data type `Packet` that has been equipped with serialization support. For making a user-defined class serializable it needs to be derived from the class `osss_serialisable_object`. In addition, it needs to use the `OSSS_IS_SERIALISABLE(_this_class_name_)` macro whose only argument is the type of the actual class.

```
1  class Packet : public osss_serialisable_object {
2  public:
3    OSSS_IS_SERIALISABLE(Packet);
4
5    // default constructor
6    OSSS_SERIALISABLE_CTOR(Packet, ());
7
8    // copy constructor
9    OSSS_SERIALISABLE_CTOR(Packet, (const Packet &pkt));
10
11   // assignment operator
12   void operator=(const Packet &pkt);
13
14   // equality operator
15   bool operator==(const Packet &pkt);
16
17   virtual void serialise() {
18     osss_serialisable_object::store_element(m_source_addr);
19     osss_serialisable_object::store_element(m_target_addr);
20     osss_serialisable_object::store_array(m_payload, 10);
21   }
22
23   virtual void deserialise() {
24     osss_serialisable_object::restore_element(m_source_addr);
25     osss_serialisable_object::restore_element(m_target_addr);
26     osss_serialisable_object::restore_array(m_payload, 10);
27   }
28
29   unsigned char get_source_addr() const;
30   void          set_source_addr(unsigned char addr);
31   unsigned char get_target_addr() const;
32   void          set_target_addr(unsigned char addr);
33   unsigned char get_payload(unsigned int index) const;
34   void          set_payload(unsigned int index,
35                             unsigned char data);
36   unsigned int  get_payload_size() const;
37
38 protected:
```

```
39   unsigned char m_source_addr;
40   unsigned char m_target_addr;
41   unsigned char m_payload[10];
42 };
```

Listing 6.29: Adding de-/serialization support to the user-defined `Packet` class

Each constructor has to be declared using the macro:
`OSSS_SERIALISABLE_CTOR(_this_class_name_, (_paramerter0_, ..., _parameterN_))`
In the virtual methods `serialise()` and `deserialise()` all attributes of the actual class that need to be serialized/de-serialized have to be registered. The registration is performed by the `store_element(...)` and the `restore_element(...)` method for scalar types, and by the `store_array(...)` and the `restore_array(...)` method for array types. These store and restore methods are provided by the `osss_serialisable_object` base class. It is very important to notice that the sequence of the store method calls in the serialise method needs to be exactly the same as the sequence of restore method calls in the deserialise method. Otherwise the resulting serialization/de-serialization behaviour is undefined. This might become hard to debug, since the `serialise()` and the `deserialise()` methods are called "automatically" whenever a serialization or de-serialization action is required.

6.6.1.3 The `osss_rmi_channel<...>` container for synthesisable OSSS-Channels

The `osss_rmi_channel<...>` is a container class for all OSSS-Channels which implement the `osss_abstract_channel` interface (e.g. buses or crossbar-switches). More simple OSSS-Channels which only implement the `osss_abstract_basic_channel` interface (e.g. point-to-point connections) need to be bidirectional in order to work inside an `osss_rmi_channel<...>` container.

```
1 typedef osss_rmi_channel<xilinx_opb_channel<false, false> > HWSWChannelType;
2 typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> > HWHWChannelType;
```

Listing 6.30: Usage of the `osss_rmi_channel<...>` container

Listing 6.30 shows the usage of an `osss_rmi_channel<...>` container in the producer/consumer example. The `HWSWChannelType` is a `xilinx_opb_channel<...>` with a least recently used scheduler and no registered grants. It allows the connection of multiple master and multiple slave components. Its data and address size is 32 bit. The `HWHWChannelType` is an `osss_rmi_point_to_point_channel<...>` that is a bidirectional point-to-point connection. Its data size is 8 bit in each direction.

The intended purpose of the `osss_rmi_channel<...>` is to separate the high-level RMI protocol from the low-level bit-accurate protocol of the channel. The channel protocol is implemented by the corresponding channel class (e.g. the `xilinx_opb_channel<...>` implements the protocol and manages the interconnection of the master and slave components as specified in the Xilinx specific implementation of the IBM On-Chip Peripheral Bus [151, 105]). All RMI protocol specific features that build on top of the channel protocol are implemented inside the `osss_rmi_channel<...>` class. The separation of RMI and the channel protocol makes it possible to design and test a channel independently from the more complex RMI protocol. The usage of well defined interfaces in both the `osss_rmi_channel<...>` and the OSSS-Channel allows to exchange one channel implementation by another. This substitution can be performed without any needs for modifying the rest of the design. This enables a convenient plug & play mechanism which allows easy exchange of physical channel implementations.

6.6.1.4 The `osss_object_socket<...>` container for Shared Objects

The `osss_object_socket<...>` container class for Shared Objects serves basically the same purpose as the `osss_rmi_channel<...>` container for OSSS-Channels. Firstly, it encapsulates the RMI protocol that is used for communication through channels (i.e. communication to the outside world) and secondly it encapsulates the method call performed on the Shared Object itself (i.e. internal communication, represents a virtual client calling a method on the Shared Objects inner class). This separation allows the designer to plug a Shared Object inside an `osss_object_socket<...>` container without the modifying any Shared Object code.

```
1 typedef osss_object_socket<osss_shared<FIFO<Packet, 10>, osss_round_robin> > BufferType;
```

Listing 6.31: Usage of the `osss_object_socket<...>` container

Listing 6.31 shows the usage of an `osss_object_socket<...>` that contains a Shared Object which contains a FIFO. Concurrent accesses are arbitrated by a round robin scheduling policy. When using this kind of object socket, the designer does not need to perform any code modifications, neither on the Shared Object nor on the `FIFO<...>` class inside of it.

6.6.1.5 The final assembly phase

In the final assembly phase we construct the top-level module containing the whole design mapped on the *Virtual Target Architecture Layer*. This involves the following steps:

1. Choose and **instantiate software processor(s)** available in the *Architecture Class Library*.
2. **Perform default mappings and substitutions**: map software tasks to software processors (single task per processor), substitute `sc_modules` by `osss_modules` and wrap Shared Objects by Object Socket containers.
3. **Instantiate RMI-Channel containers**. Choose OSSS-Channels from the *Architecture Class Library* or implement a synthesizable OSSS-Channel for your special needs. **Plug an OSSS-Channel into each RMI-Channel container**.
4. **Perform logical and physical bindings**: The *logical binding* represents the port to interface binding from the Application Layer Model. The *physical binding* describes the connection of the architecture building blocks (like processors, object sockets, hardware modules) to the RMI-Channel containers.

As one can see from these four steps, it does not require any changes of the behaviour of any software tasks or any hardware modules from the *Application Layer Model*. Nevertheless, after a profiling run of the Application Model mapped on the Virtual Target Architecture some changes or optimizations of the application's behaviour might become apparent. These changes can be performed separately on the *Application Layer Model* without affecting the chosen target architecture.

Listing 6.32 highlights all modifications in the top-level design of the producer/consumer example during Application to Virtual Target Architecture Layer mapping. A graphical representation of the design described in Listing 6.32 can be found in the lower part of Figure 6.36.

In the first part of Listing 6.32 two different kinds of RMI-Channels are defined (see Listing 6.30). The `HWSWChannelType` is used for communication between the Producer (software) and the Shared Object (hardware). The `HWHWChannelType` is used for communication between the Consumer (hardware) and the Shared Object. This RMI-Channel definition is followed by the definition of the bounded Packet FIFO (`BufferType`) that is a Shared Object plugged into an `osss_object_socket<...>` (see Listing 6.31).

```
1  #define OSSS_GREEN
2  #include "osss.h"
3
4  class Top : public osss_system {
5  public:
6
7    sc_in<bool>    pi_bClk;
8    sc_in<bool>    pi_bReset;
9
10   typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
11     HWSWChannelType;
12   typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
13     HWHWChannelType;
14
15   typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
16                                          osss_round_robin> > BufferType;
17
```

```
18    Producer*           m_Producer;
19    HWSWChannelType*  m_Channel1;
20    BufferType*         m_Buffer;
21    HWHWChannelType*  m_Channel2;
22    Consumer*           m_Consumer;
23
24    xilinx_microblaze* m_Processor;
25
26    sc_signal<Packet> ms_Packet;
27
28    Top(sc_core::sc_module_name name) : osss_system(name) {
29      m_Channel1 = new HWSWChannelType("m_Channel1");
30      m_Channel1->clock_port(pi_bClk);
31      m_Channel1->reset_port(pi_bReset);
32
33      m_Channel2 = new HWHWChannelType("m_Channel2");
34      m_Channel2->clock_port(pi_bClk);
35      m_Channel2->reset_port(pi_bReset);
36
37      m_Buffer = new BufferType();
38      m_Buffer->clock_port(pi_bClk);
39      m_Buffer->reset_port(pi_bReset);
40      m_Buffer->bind(*m_Channel1);
41      m_Buffer->bind(*m_Channel2);
42
43      // this is a software task
44      m_Producer = new Producer("m_Producer");
45      m_Producer->clock_port(pi_bClk);
46      m_Producer->reset_port(pi_bReset);
47      m_Producer->output(*m_Buffer);
48
49      m_Processor = new xilinx_microblaze("m_Processor");
50      m_Processor->clock_port(pi_bClk);
51      m_Processor->reset_port(pi_bReset);
52      // this port binds the processor to its bus (m_Channel1)
53      m_Processor->rmi_client_port(*m_Channel1);
54      // here the above software task is added to this processor
55      m_Processor->add_sw_task(m_Producer);
56
57      m_Consumer = new Consumer("m_Consumer");
58      m_Consumer->pi_bClk(pi_bClk);
59      m_Consumer->pi_bReset(pi_bReset);
60      m_Consumer->po_Packet(ms_Packet);
61      m_Consumer->input(*m_Channel2, *m_Buffer);
62    }
63  };
```

Listing 6.32: Modifications on the top-level module of the consumer/producer example

In the constructor of the top-level module `Top` both channels `m_Channel1` and `m_Channel2` are instantiated and bound to the global clock and reset signals.

On the *Application Layer*, the producer and the consumer client were both directly bound to the Shared Object. Due to the communication refinement, by inserting RMI-Channels between the producer and the Shared Object as well as between the consumer and the Shared Object the bindings of the `m_Buffer`, `m_Producer` and `m_Consumer` need to be adapted. The producer and the Shared Object are bound to the same channel (`m_Producer` and `m_Buffer` are both bound to `m_Channel1`) and the consumer and the Shared Object are also bound to the same channel (`m_Consumer` and `m_Buffer` are both bound to `m_Channel2`).

When an `osss_object_socket<...>` is connected to a shared bus as a slave module an address map for that slave becomes necessary. An address map consists of a base and high address that specify the address range a slave component is sensitive to. When a master drives the address lanes inside the OSSS-Channel the slave whose address range includes this address gets active and serves the masters request. Although it is possible to specify the address maps manually we suggest the designer not to do so unless he knows exactly what he is doing. In the normal case all address maps are calculated and internally assigned by the `osss_rmi_channel<...>` automatically.

When binding a port of type `osss_port<osss_rmi_if<...> >` to an `osss_rmi_channel<...>` on the *Virtual Target Architecture Layer* the binding information of the *Application Layer* is retained. A second parameter of the `operator()` of the `osss_port<osss_rmi_if<...> >` class was introduced for that purpose. Having a look at the code in Listing 6.32 the output port of the producer is bound to the RMI-Channel and to the object that is plugged into the `osss_object_socket<...>` (i.e. the Shared Object itself). The need for retaining this binding information from the *Application Layer* is at least necessary for the simulation. Since, by the first method call performed on an `osss_port<osss_shared_if<...> >` the corresponding process doing this call is registered at the Shared Object. The same behaviour has to be retained after mapping the application to the *Virtual Target Architecture* and thus using `osss_port<osss_rmi_if<...> >`.

Until now OSSS is only capable of dealing with a single clock domain per system. This restriction can be exploited for writing more concise top-level modules. Listing 6.33 shows how to use the static `osss_global_port_registry` class to register the global clock and reset signal.

Each component of the *Virtual Target Architecture Layer* provides a clock and reset interface that defines a `clock_port` and a `reset_port`, both of type `sc_in<bool>`. The designer can either decide to perform a manual binding of these ports, or to omit the binding which results in an automatic binding to the globally registered clock and reset port.

When mapping a design from the *Application* to the *Virtual Target Architecture Layer* the designer has to take care to replace any `sc_module` by an `osss_module`. All other architecture building blocks like processors, object sockets, channels and memories are already capable of the automatic clock and reset port binding.

```
1  #define OSSS_GREEN
2  #include "osss.h"
3
4  class Top : public osss_system {
5  public:
6
7    sc_in<bool>    pi_bClk;
8    sc_in<bool>    pi_bReset;
9
10   typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
11     HWSWChannelType;
12   typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
13     HWHWChannelType;
14
15   typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
16                                          osss_round_robin> > BufferType;
17
18   Producer*          m_Producer;
19   HWSWChannelType*   m_Channel1;
20   BufferType*        m_Buffer;
21   HWHWChannelType*   m_Channel2;
22   Consumer*          m_Consumer;
23
24   xilinx_microblaze* m_Processor;
25
26   sc_signal<Packet> ms_Packet;
27
28   Top(sc_core::sc_module_name name) : osss_system(name) {
29     // register clock and reset ports and make them global
30     osss_global_port_registry::register_clock_port(pi_bClk);
31     osss_global_port_registry::register_reset_port(pi_bReset);
32
33     m_Channel1 = new HWSWChannelType("m_Channel1");
34     m_Channel2 = new HWHWChannelType("m_Channel2");
35
36     m_Buffer = new BufferType();
37     m_Buffer->bind(*m_Channel1);
38     m_Buffer->bind(*m_Channel2);
39
40     // this is a software task
41     m_Producer = new Producer("m_Producer");
42     m_Producer->output(*m_Buffer);
43
```

```
44      m_Processor = new xilinx_microblaze("m_Processor");
45      // this port binds the processor to its bus (m_Channel1)
46      m_Processor->rmi_client_port(*m_Channel1);
47      // adds the above software task to this processor
48      m_Processor->add_sw_task(m_Producer);
49
50      //CAUTION: Make shure the Consumer has been chagned from sc_module
51      //         to osss_module. Otherwise automatic clock and reset port
52      //         binding does not work!
53      m_Consumer = new Consumer("m_Consumer");
54      m_Consumer->po_Packet(ms_Packet);
55      m_Consumer->input(*m_Channel2, *m_Buffer);
56    }
57  };
```

Listing 6.33: The top-level module from Listing 6.32 with global clock and reset port bindings

6.6.2 Architecture Exploration

In the previous sections we have shown how to map an *Application Layer* model to a *Virtual Target Architecture Layer* model. After presenting the basic mapping steps this section serves as a starting point for a simple top-down architecture exploration. During this section we present two different communication mappings of the producer/consumer example and discuss some of the profiling results generated from model execution. Moreover, the presented example demonstrates the flexibility to quickly change the target architecture mapping.

In OSSS, communication links (port to interface bindings) are mapped onto communication resources of the *Virtual Target Architecture Layer*, implemented as OSSS-Channels. The provided flexibility for the designer is very high, since these channels can differ in connection topologies (ranging from a point-to-point, over a shared bus to a full featured $N \times N$ crossbar-switch), bit sizes and in their communication protocols. Figure 6.37 illustrates two different mapping alternatives of the producer/consumer example introduced in Figure 6.14.

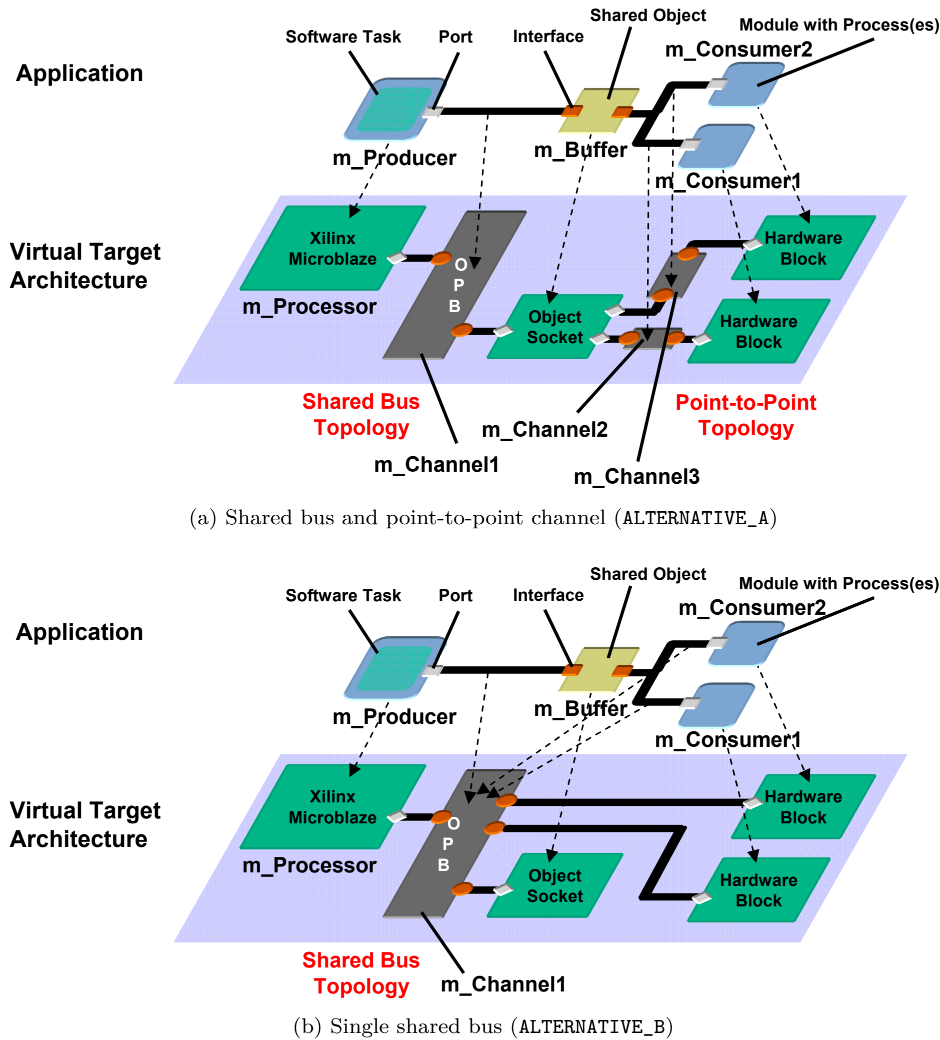
In all mappings shown in Figure 6.37 the producer software task has been mapped to a Xilinx MicroBlaze^{TM}processor. It would have also been possible to map this task to any other processor available in the *Architecture Class Library*. One limitation of the presented OSSS refinement methodology is that only a single Software Task can be mapped onto each software processor. Current work that removes this restriction and allows modeling of software multitasking can be found in [48, 23, 17].

Since the MicroBlaze processor has a built-in OPB interface we have connected it to a Xilinx OPB channel. However, it is also possible to connect any other channel to the MicroBlaze processor, but this would imply the use of a bridge or protocol converter. For the sake of simplicity we have chosen the OPB in this example.

One of the main goals of the OSSS methodology is to provide a seamless synthesizable communication refinement for hardware/software systems. For the producer/consumer example this means the behaviours inside the producer software task and the consumer hardware modules are *not* affected by the mapping and communication refinement. In particular, the high-level communication mechanism (of the *Application Layer*) using method calls on user-defined interfaces persists after mapping on the *Virtual Target Architecture Layer*.

The OSSS-Channels on the VTA Layer implement a synthesizable signal-level communication using architecture specific topologies and communication protocols, like the OPB. To retain the user-defined method calls from the application model we need a concept to translate them to these low-level communication resources. This kind of translation is usually performed by a network protocol stack defining several protocol layers that abstract from the underlying physical communication resource. In the OSSS methodology we call this concept **R**emote **M**ethod **I**nvocation (RMI). It enables the call of a method of a remote object through a physical (i.e. signal-level) connection. More details about the OSSS RMI concept and protocol stack have been presented in Section 6.5.2.

Each communication link from the *Application Layer* can be mapped to any `osss_rmi_channel<...>` container (denoted by `m_Channel1 - 3` in Figure 6.37). They serve as



(a) Shared bus and point-to-point channel (ALTERNATIVE_A)

(b) Single shared bus (ALTERNATIVE_B)

Figure 6.37: Different mapping alternatives of the producer/consumer application [44]

wrappers for the OSSS-Channels, which implement the physical structure and the behaviour of communication protocols like buses (e.g. OPB) or point-to-point connections. The purpose of the RMI-Channels is the provision of a specific RMI interface and the translation of the OSSS-RMI protocol to the physical channel protocol.

Listing 6.34 shows the refined and mapped top-level design of the producer/consumer example on the Virtual Target Architecture Layer. The two different communication mapping alternatives from Figure 6.37 are marked with `ALTERNATIVE_A/B` pre-processor definitions. Another main difference between the Application and the Virtual Target Architecture Layer top-level design is the use of the `xilinx_system` base class (line 8). It serves two purposes: Firstly, it marks the top-level entity of the design used for synthesis. Secondly, it adds a "hook" to analyze the structure of the top-level design and generates target specific architecture definition and configuration files. When using the `xilinx_system`, configuration files for the Xilinx Platform Studio are generated during the SystemC elaboration phase (see Section 7.5).

At the beginning two different channel types are defined: the Xilinx OPB (`Bus_Ch_t`) and the point-to-point (`P2P_Ch_t`) channel with a client bit width of 8 (used by the initiator of the communication) and a server bit width of 32 (used by the target of the communication, i.e. Shared Objects). In the constructor (line 34) all channels are instantiated and bound to clock and reset. Depending on the mapping alternative the buffer Shared Object is bound to the physical connected channels using the `bind` method of the Object Socket (e.g. line 50). The producer Software Task is instantiated right before the Xilinx MicroBlaze which is bound to

the OPB channel by its `rmi_client_port` (line 62). The `add_sw_task` method is used to map the producer Software Task to the MicroBlaze (line 63). After mapping the producer to the processor all communications using the `output` port of the Software Task are performed through the connected OPB channel. The binding of the `input` port of the consumer hardware modules gets a second parameter (line 71 & 72) on the Virtual Target Architecture Layer. The first one defines the physical binding to a communication channel. The second parameter defines the same logical binding to the Shared Object as on the Application Layer.

```
1  #define OSSS_GREEN // Virtual Target Architecture Layer Model
2  #include <osss.h>
3  #include "Packet.hh"
4  #include "FIFO.hh"
5  #include "Producer.hh"
6  #include "Consumer.hh"
7
8  class Top : public xilinx_system {
9   public:
10    sc_in<bool> clk, reset;
11
12    typedef
13    osss_rmi_channel<xilinx_opb_channel<> > Bus_Ch_t;
14
15    typedef
16    osss_rmi_channel<
17      osss_point_to_point_channel<8, 32> > P2P_Ch_t;
18
19    typedef
20    osss_object_socket<
21     osss_shared<FIFO<Packet, 10>, osss_round_robin> > Buffer_t;
22
23   protected:
24    Bus_Ch_t *m_Channel1;
25    P2P_Ch_t *m_Channel[2];
26
27    Producer *m_Producer;
28    Buffer_t *m_Buffer;
29    Consumer *m_Consumer[2];
30
31    xilinx_microblaze *m_processor;
32
33   public:
34    Top(sc_module_name name) : xilinx_system(name) {
35      m_Channel1 = new Bus_Ch_t("m_Channel1");
36      m_Channel1->clock_port(clk);
37      m_Channel1->reset_port(reset);
38  #ifdef ALTERNATIVE_A
39      m_Channel[0] = new P2P_Ch_t("m_Channel2");
40      m_Channel[1] = new P2P_Ch_t("m_Channel3");
41      for(unsigned int i=0; i<2; ++i) {
42        m_Channel[i]->clock_port(clk);
43        m_Channel[i]->reset_port(reset);
44      }
45  #endif
46
47      m_Buffer = new Buffer_t("m_Buffer");
48      m_Buffer->clock_port(clk);
49      m_Buffer->reset_port(reset);
50      m_Buffer->bind(*m_Channel1);
51  #ifdef ALTERNATIVE_A
52      m_Buffer->bind(*m_Channel[0]);
53      m_Buffer->bind(*m_Channel[1]);
54  #endif
55
56      m_Producer = new Producer("m_Producer");
57      m_Producer->output(*m_Buffer);
58
59      m_Processor = new xilinx_microblaze("m_Processor");
60      m_Processor->clock_port(clk);
61      m_Processor->reset_port(reset);
62      m_Processor->rmi_client_port(*m_Channel1);
```

```
63      m_Processor−>add_sw_task(m_Producer);
64
65      m_Consumer[0] = new Consumer("m_Consumer0");
66      m_Consumer[1] = new Consumer("m_Consumer1");
67      for(unsigned int i=0; i<2; ++i) {
68        m_Consumer[i]−>clock_port(clk);
69        m_Consumer[i]−>reset_port(reset);
70  #ifdef ALTERNATIVE_A
71        m_Consumer[i]−>input(∗m_Channel[i], ∗m_Buffer);
72  #else // ALTERNATIVE_B
73        m_Consumer[i]−>input(∗m_Channel1, ∗m_Buffer);
74  #endif
75      }
76    }
77  };
```

Listing 6.34: Top-Level module of the producer/consumer example on the VTA Layer

To demonstrate the impact of different communication mappings for the producer/consumer example we have performed a packet throughput measurement. The measurement has been performed on the Application Layer Model (ref. Figure 6.14) and the two different Virtual Architecture Models (ref. Figure 6.37). Table 8.12 shows the results of a simulation with 2000 produced packets at a clock frequency of 100.0 MHz. The simulation time is the duration of the entire simulation run measured on the same reference workstation.

Implementation Model	Simulation Time[a] [s]	Packet Throughput [Packets/s]
Application Layer	0.2	12 512 512.5
Virtual Target Architecture Layer		
`ALTERNATIVE_A`: OPB & P2P channels	6.4	1 853 705.6
Virtual Target Architecture Layer		
`ALTERNATIVE_B`: OPB channel only	5.8	848 413.9

[a] Intel(R) Pentium(R) 4 CPU 3.00GHz

Table 6.3: Simulation results of the different producer/consumer models

The *Application Layer Model*'s simulation time is the shortest while the measured packet throughput is the highest. This result is not surprising since this layer abstracts from all communication details. The simulation runs of both *Virtual Target Architecture Models* take much longer (about a factor of 30) because they perform a cycle accurate simulation of the physical communication. A more interesting result is the significant lower packet throughput of mapping alternative B compared to mapping alternative A. Since alternative B uses only a single OPB channel we observe lots of bus contention that slows down the packet throughput dramatically.

This hierarchical RMI-Channel enables the use of arbitrary protocols and physical connection topologies encapsulated by OSSS-Channels. This "plug-and-play" mechanism is very useful during communication architecture exploration or for altering design decisions late in the design flow. For instance the MicroBlaze and its associated OPB channel can be easily exchanged by an ARM processor connected to an AMBA AHB channel.

For the refinement of the communication mapping the designer only needs to instantiate an OSSS-RMI-Channel container and plug an OSSS-Channel taken from the architecture class library into it. Due to the dynamic channel generation and adaptivity, software processors, object sockets and hardware modules can directly be bound to OSSS-RMI-Channel containers. They completely abstract from the chosen channel physical behaviour including protocol and topology.

6.7 Summary

This summary gives an overview and comparison of passive and active modeling elements of the OSSS simulation library and describes the mapping possibilities from the Behavioral to the

Application and from the Application to the Virtual Target Architecture Layer. This summary closes with a review of requirements, concerning the simulation model, from Chapter 2.

6.7.1 Passive Modeling Elements

	Plain Object	Event	Shared Variable<X>	Shared Object<Y>	Adapter Socket<Z>
Layer	Behavioral/ Application	Behavioral	Behavioral	Application	Application
Object Type	Value	Entity	Entity	Entity	Entity
Usage	Inline	Shared	Shared	Exclusive/ Shared	Exclusive/ Shared
Assign-/Copyable	Yes/Yes	No/No	Yes/No	No/No	No/No
Serializable	Yes	No	No	No	No
Supports Inheritance	Yes	No	No	No	No
Access arbitration	No	No	No	Yes	Yes
Guarded access	No	No	No	Yes	Yes
Provides interface(s)	No	Yes[a]	Yes[a]	Yes[b]	Yes[b]
Provides port(s)	No	No	No	No	Yes
Requires clock	No	No	No	Yes	Yes
Requires reset	No	No	No	Yes	Yes
Nesting allowed	Yes	No	No	No	No

with X ∈ { Basic, Plain Object }, Y ∈ { Plain Object }, Z ∈ { Shared<X> }.
[a] Fixed interface(s).
[b] User-defined interface(s).

Table 6.4: Overview of passive OSSS modeling elements

Table 6.4 gives an overview of the passive OSSS modeling elements presented in the this chapter. In OSSS we distinguish between value and entity object types in the following way:

Value object types have an implicit location. They do not describe an own data path, but are embedded in the calling or owning thread. Therefore, member functions of value objects are executed inline to the thread that uses them. Like all objects they are initializable which usually happens during construction (the constructor is responsible for that). Moreover they are assignable, copyable and serializable. Assignable means that any other value object of the same type can be assigned. Copyable means initialization through assignment. Serialization is a special case of copy that enables to write the state of a value object to a bit vector representation that can either be stored and restored from a memory or can be used to send it through a physical channel, like OSSS-Channels using the OSSS-RMI protocol.

Entity objects can not be copied and sent via RMI-Channels. They have an explicit location with an exclusive data path. Member function calls on entity objects are therefore not inlined in the caller thread. They are executed in their own thread on their own data path. Of course entity objects are initializable like value objects.

The different object types usually correspond to three different usage patterns: inline, exclusive and shared:

Inline usage pattern means that data and behaviour of an object is embedded into the owning thread. Since this is the strongest from of exclusiveness (used by a single process only) usage of guards is not allowed.

exclusive usage pattern means that an object is accessed by a single process. This does not require any arbitration (like for the inline pattern) guards are not allowed. The main difference between inline and exclusive lies in the structural separation of caller and callee.

In the exclusive usage pattern the object has its own dedicated data path apart from the data path of the caller.

shared usage pattern is a special case of the exclusive one (each Shared Object has all properties of an Exclusive Object). In contrast to an exclusive object a shared object is capable of being accessed by more than a single process. To avoid race condition arbitration through guarded methods (some kind of semaphore) and/or an explicit arbiter is necessary.

Table 6.4 shows that value objects are used inline and entity objects are used exclusive or shared in OSSS.

Serialization is the process of saving an object onto a storage medium (such as a file, or a memory buffer) or to transmit it across a network connection link in binary form. The series of bytes or the format can be used to re-create an object that is identical in its internal state to the original object (actually, a clone). In OSSS plain objects that need to be send through RMI-Channels need to be serializable. In some programming languages (e.g. Java), **transient** is a keyword used as a field modifier. In OSSS all serializable objects also support transience. When a field is declared transient, it would not be serialized even if the class to which it belongs is serialized. In OSSS transience is only supported for serializable objects. Fields that are not explicitly serialized or de-serialized in the `serialise()` and `deserialse()` methods of a serializable object are transient and will not be sent through an RMI-Channel. When the object is restored/de-serialized the transient fields are initialized with their default values (as specified in the constructor).

As we have described in Section 6.5 communication in OSSS is expressed statically through Port-Interface-Bindings. These binding constitute so-called communication links that are mapped and refined to OSSS-RMI Channels at the VTA Layer.

6.7.2 Active Modeling Elements

Table 6.5 gives an overview of the available active OSSS modeling elements. In contrast to passive modeling elements from the previous section, active modeling elements act as initiators, since they own a thread of control.

	Behavior `osss_behaviour`	Hardware Module (`osss_module`)	Software Task (`osss_sw_task`)
Layer	Behavioral	Application	Application
Object Type	Entity	Entity	Entity
Usage	Active	Active	Active
Process Type	`SC_THREAD`[a] & `SC_CTHREAD`[a]	`SC_CTHREAD` & `SC_METHOD`	`SC_CTHREAD` & `SC_THREAD`
No. of threads	1 - N	0 - N	1
Assign-/Copyable	No	No	No
Supports Inheritance	Yes	Yes	Yes
Requires clock	Yes[b]	Yes	Yes[c]
Requires reset	Yes[b]	Yes	Yes[c]
OSSS ports	Yes	Yes	Yes
Signal ports	Yes	Yes	No
Nesting allowed	Yes[d]	Yes	No

[a] The process type is decided at the root Behavior. For modeling of untimed systems, using delta notifications only, `SC_THREAD` is required. For using Behaviors to model clock cycle accurate, `SC_CTHREAD` is required.
[b] When using the `SC_CTHREAD` for modeling on clock cycle granularity, the root Behavior need to be bound to clock and reset.
[c] Only on Application Layer. After adding SW Task to CPU no clock and reset is needed.
[d] Sub-Behaviors, called composite Behaviors, can be composed in sequential (SEQ), finite state machine (FSM), parallel (PAR) or pipelined (PIPE) execution order.

Table 6.5: Overview of active OSSS modeling elements

6.7.3 Mapping and Refinement

Table 6.6 gives and overview of the OSSS Behavioral Layer (BL) to Application Layer (AL) mapping and refinement possibilities. Leaf Behaviors and hierarchical Behaviors b with $|par_set(b)| = 1$ using sequential (SEQ) and finite state machine (FSM) composition can be mapped to HW Modules and SW Tasks. Hierarchical Behaviors b with $|par_set(b)| > 1$ using parallel (PAR) and pipeline (PIPE) compositions are mapped to the Application Layer using the PAR (see Listing C.6) and PIPE (see Listing C.7) Shared Objects. Events cannot be replaced by Shared Objects directly. Instead, Shared Object replacements for all pre-defined OSSS Behavioral Channels are provided: *Queue* (see Listing C.4), *Handshake* (see Listing C.2) and *Double Handshake* (see Listing C.3). Shared and Piped Variables from the OSSS Behavioral Layer are replaced by *Shared Variable* (see Listing C.1) and *Piped Variable* (see Listing C.5) Shared Objects.

AL Elements **BL Elements**	**HW Module**	**SW Task**	**Shared Object**
Behavior	Leaf Behavior Hierarchical Behavior	Leaf Behavior Hierarchical Behavior	PAR composition PIPE composition
Event	–	–	No (Double) Handshake Queue
Shared Variable	–	–	Yes

Table 6.6: Overview of OSSS Behavioral Layer (BL) to Application Layer (AL) mapping & refinement possibilities

VTAL Elements **AL Elements**	**SW Processor**	**HW Block**	**Memory**	**Channel**
HW Module	–	change `sc_module` to `osss_module`	–	–
SW Task	only single Task per processor[a]	–	–	–
Plain Object	inlined	inlined	storage[b]	–
Shared Object	Not yet[c]	wrapped by Object Socket	shared state[d]	–
Adapter Socket	–	wrapped by Object Socket	–	–
Communication Link	–	–	Not yet[d]	RMI

[a] Multitasking is not supported here. An extension to support multitasking can be found in [48, 23, 17].
[b] Value objects should be mappable to dedicated memories. Read-Modify-Write semantics should be supported.
[c] Along with our works on the support of multitasking we think about the support of Shared Objects for inter-task communication.
[d] The use of dedicated memories for the mapping of communication links can be used for the implementation of a call-by-reference mechanism.

Table 6.7: Overview of OSSS Application Layer (AL) to Virtual Target Architecture Layer (VTAL) mapping & refinement possibilities

Table 6.7 gives an overview of the OSSS Application Layer (AL) to Virtual Target Architecture

Layer (VTAL) mapping and refinement possibilities. Until now we have a strict separation of hardware and software. Hardware Modules are refined into synthesizable behavioral RTL descriptions and Software Tasks are executed on a software processor. An extension of the presented modeling elements supports multitasking [48, 23, 17]. Thus, enabling the mapping of more than a single Software Task to a processor, including the mapping of Shared Objects to software processors and shared memory. Plain Objects can either be used in Software Tasks or in Hardware Modules. In both cases they become inlined to the thread of control. The state of a Plain Object can be mapped to a dedicated memory. However until now it is not possible to modify attributes of memory mapped objects directly. The support of a Read-Modify-Write mechanism for the purpose of efficient attribute manipulation will be part of a future release of the OSSS library.

Until now Shared Objects can be implemented in dedicated hardware, thus limiting the mapping possibilities and flexibility (and requiring the usage of FPGA platforms of a full custom ASIC design). This is performed by wrapping them with a so-called Object Socket. In the extension to support SW multitasking, Shared Objects can also be used inter-task communication.

Communication Links from the OSSS Application Layer (constituted by port to interface bindings) are mapped onto physical communication channels on the OSSS Virtual Target Architecture Layer. With the support of OSSS RMI-Channel containers a generic decoupling from application-specific method-based communication to architecture specific signal-level communication is given. The use of dedicated memories for the mapping of communication links can be used for the implementation of a call-by-reference mechanism. But this is also future work.

6.7.4 Review of Goals

In Table 6.8 a review of the related goals from Chapter 2 is given.

Table 6.8: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled)

ID	Goal	Status	Comment
G1	Integration of synthesis tool and simulation infrastructure into Eclipse CDT Framework	●	Since OSSS is based on C++ and integration into the Eclipse C/C++ Development Tooling (CDT) Framework is possible. Both, the simulator and the synthesis tool *Fossy* have been successfully integrated, see Chapter G.
G2	Introduce a notion of time for the SW parts	●	Estimated Execution Time (EET) blocks, as described in Section 6.4.3.3, enable timing annotation of Software Tasks. Required Execution Time (RET), as described in Section 6.4.3.4, enable dynamic run-time checks of software timing requirements.
M1	Single modeling language to describe HW and SW	●	OSSS, as introduced in Section 6.2 covers the description of HW and SW. At the Application Layer, SW is described by Software Tasks (see Section 6.4.3) and HW is described by Hardware Modules (see Section 6.4.5).
M2	SystemC approach	●	OSSS is based on SystemC as described in Section 6.2.

continued on next page

Table 6.8: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
M3	Executable Specification and HW/SW partitioned models	●	The OSSS simulation model covers the untimed specification level modeling (OSSS Behavioural Layer, see Section 6.3), timed HW/SW partitioned modeling (OSSS Application Layer, see Section 6.4) and timed execution platform modeling (OSSS Virtual Target Architecture Layer, see Section 6.5).
M4	Synthesizable HW/SW partitioned model	●	An OSSS Application Layer model (which describes a HW/SW partitioned model) mapped to an OSSS Virtual Target Architecture Layer model is synthesizable with the prototypical synthesis tool *Fossy*, see Chapter 7.
M10	Possibility to write hardware modules at RT-level	●	HW Modules at the Application Layer (see Section 6.4.5) are described at behavioral RT (using `SC_CTHREAD`s). For more details see Chapter F.
M12	Consideration of (real-)time constraints	◖	The combination of EETs and RETs can be used to specify real-time constraints. Currently RETs can only be used within one Actor. This modeling restriction restricts the expressiveness of advanced (real-)time constraints, e.g. end-to-end deadlines.
M14	Integration of IP components	●	IP components can be integrated using IP Component wrapper modules. Signal based communication with Hardware Modules and Shared Objects plugged into Adapter Sockets is supported.
A1	Debugging on all levels of abstraction	◖	Debugging of OSSS simulation models is not directly supported by the methodology or simulation library. But since OSSS builds on top of SystemC and C++, common waveform visualization and debugging tools can be used.
A3	Basic timing properties shall be reflected by the simulation	●	At the Application Layer timing properties of Software Tasks are represented by EET blocks and timing properties of Hardware Modules are represented by SystemC `wait()` statements. At the Virtual Target Architecture Layer timing properties of the RMI protocol and the on-chip communication resources (bus, point-to-point) are added.
A4	Combine models of different levels of abstraction in a single simulation	◖	In OSSS Behavioural Layer and Application Layer modeling elements can be simulated together. The enables stepwise refinement from an untimed to a timed simulation model. Application and Virtual Target Architecture Layer modeling elements cannot be mixed (i.e. all Application Layer elements need to be mapped to their corresponding Virtual Target Architecture elements). At the Application Layer it is possible to combine timing approximate models with cycle-accurate models.

continued on next page

Table 6.8: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
A5	Consideration of IP components in the simulation	◖	SystemC RTL IP components can be integrated at the Application Layer. VDHL or Verilog RTL IP components can only be integrated when using a SystemC and VHDL or Verilog co-simulation environment (e.g. ModelSim).

CHAPTER 7

Synthesis[1]

7.1 Introduction

This chapter presents the OSSS synthesis flow. As defined in Section 5.1, synthesis in OSSS is the transformation of an Application Layer model mapped to a Virtual Target Architecture Layer model into an Implementation Layer model. In the Y-Chart in Figure 5.1 this is the step from (D) to (F). Looking to the *backend* of the design flow presented in Figure 5.3, the synthesis step consists of hardware, software and hardware/software interface synthesis.

Figure 7.1 shows the OSSS synthesis flow for the producer/consumer example introduced in Chapter 6. For synthesis we incorporate the behavior as defined by the *Application Layer Model*, the allocated hardware resources of the *Virtual Target Architecture Model* and the mapping information between them.

During the *Architecture Extraction* process the Virtual Target Architecture Model is analyzed and separated into software and hardware partitions. The *software partition* in Figure 7.1 consists of the Xilinx MicroBlaze together with its associated SW Task and the OPB channel. The hardware partition is made up of a single Object Sockets containing the buffer Shared Object, the two consumer hardware modules and two point-to-point channels.

Since our prototypical synthesis backend flow uses the Xilinx Platform Studio and the Embedded Development Kit (EDK) we are generating vendor-specific architecture definition files. These are the MSS (**M**icroprocessor **S**oftware **S**pecification) and MHS (**M**icroprocessor **H**ardware **S**pecification) file that are used for the creation of an EDK project.

For the implementation of the *Software Partition* we use the MicroBlaze, the OPB and an OPB DDR-RAM controller from the Xilinx IP core library. The Software Task is extracted from the *Application Layer Model*, cross-compiled and linked against a specific OSSS software driver library. It enables the communication with the Shared Objects using the OSSS RMI protocol.

The entire *Hardware Partition* is transformed from SystemC/OSSS to synthesizable VHDL code by our high-level synthesis tool *Fossy* (**F**unctional **O**ldenburg **S**ystem **SY**nthesiser). The resulting VHDL code is inserted into the generated EDK project and further processed by the Xilinx Synthesis Tool (XST) or other third-party RTL synthesis tools. After a mapping (MAP) and place & route (PAR) step the entire design can be downloaded to a Xilinx FPGA prototyping board.

The organization of this chapter is described in the following section using Figure 7.2:

① The input for synthesis is the Application and Virtual Target Architecture Layer model, as described in Chapter 6.

② Section 7.3 describes the basic parsing and intermediate representation of the input model.

[1] This section is based on own previous work [44].

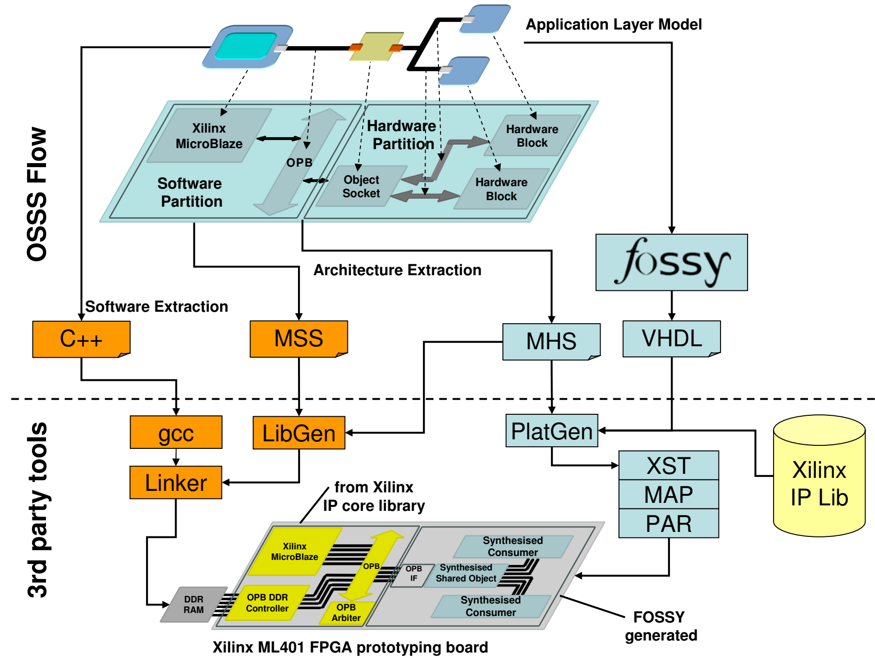


Figure 7.1: OSSS synthesis flow overview for the producer/consumer example

③ Section 7.4 describes the representation of the Virtual Target Layer modeling elements for the chosen Xilinx FPGA platform. This is followed by the description of the platform synthesis process interfacing the specific Xilinx tools in Section 7.5.

④ Section 7.6 describes the library-based software synthesis approach.

⑤ Section 7.7 describes the custom hardware synthesis and Section 7.8 the Shared Object synthesis.

⑥ Section 7.9 describes the integration with the Xilinx back-end synthesis and compilation tools to implement the specified system on the FPGA platform.

This chapter closes with a summary (Section 7.10) and a review of all synthesis related requirements from Chapter 2.

7.2 Overall Flow

We now provide a deeper insight into our synthesis flow. Figure 7.2 shows the main phases of the OSSS synthesis flow for embedded hardware/software systems. The system design takes place inside the OSSS *Design Environment* ① consisting of a single *Application Model*, (possibly) multiple *Virtual Target Architecture (VTA) Model* alternatives and a mapping relation between them. For the simulation and evaluation of these different systems the OSSS simulation library is used. The entire synthesis flow can be split into two phases:

Front-end: Describes the target platform and hardware independent part of the overall synthesis flow, and can be subdivided into the following steps:

1. OSSS *design parsing and intermediate representation* ② (see Section 7.3) is performed by a C++ front-end. The Application Layer design is represented as Abstract Syntax Tree (AST). Hardware, software and hardware/software interface synthesis work on this AST representation of the OSSS design.
2. *Design elaboration and architectural context extraction* ③ (see Section 7.5) performs analysis of the Application Layer, Virtual Target Architecture Layer, and the mapping. As a result, the hardware platform configuration, containing both, IP components

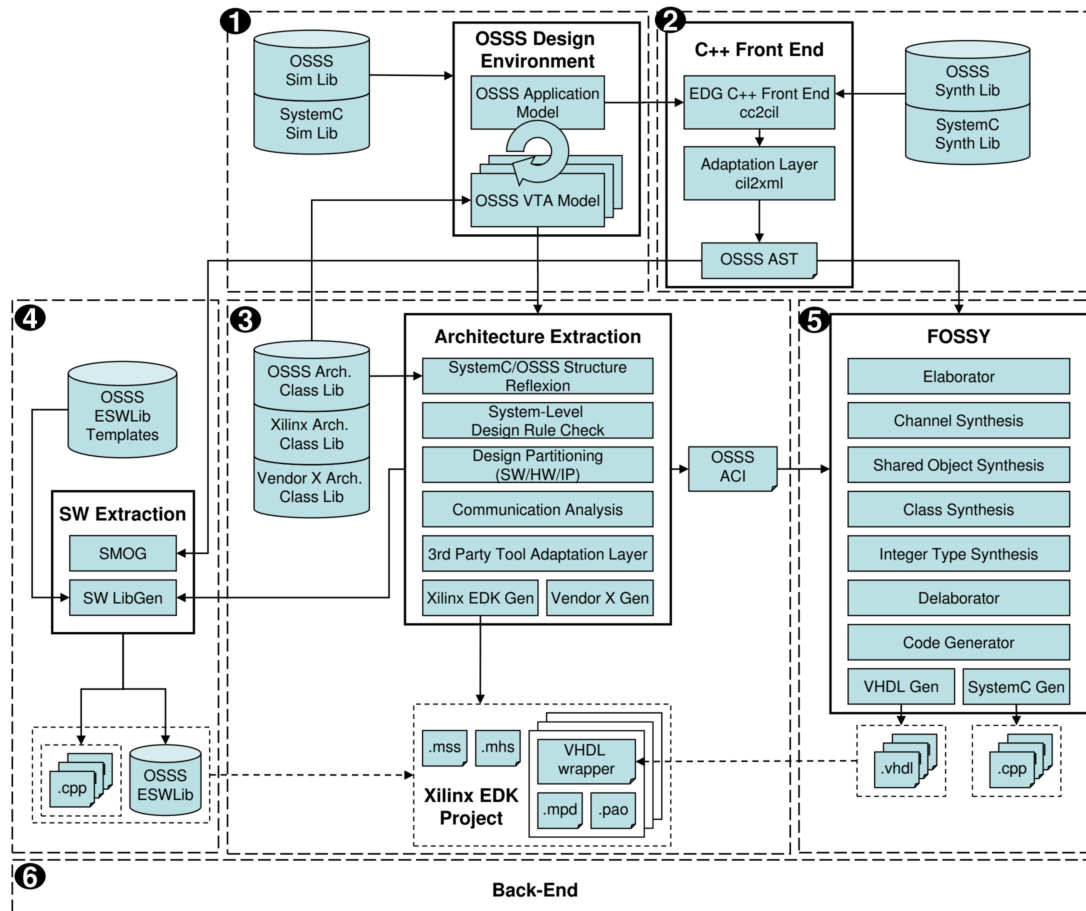


Figure 7.2: Overview of the OSSS synthesis flow front-end

and custom hardware blocks is represented in an intermediate format. It describes the configuration of each IP component and custom hardware block, and the connection via point-to-point channels or shared buses. For shared buses, the address layout of the overall system is generated, based on the register interface information (required width and depth) of each IP component.

3. *Software extraction and configuration* ④ (see Section 7.6) extracts Software Tasks from the Application Layer model and integrates them each into a self-contained cross-compilable structure. Before Software Tasks have been part of the Application Layer model. Now each software task is prepared to be executed on its own target processor. For communication with peripherals the software driver library is configured with the target addresses assigned during design elaboration and architectural context extraction. For the RMI calls to Shared Objects the RMI ports are configured with the base address of the register interface of the Shared Object, also assigned during design elaboration and architectural context extraction.

4. *High-level synthesis of the user-defined hardware part and the hardware/software interface of the design* ⑤ (see Section 7.7 and Section 7.8) transforms all custom hardware blocks, Shared Objects, and custom designed hardware to hardware communication channels (e.g. point-to-point communication) into synthesizable VHDL code. Shared Objects that have a connection to a bus get a generic memory interface that can be attached to different bus slave front-ends. For each supported system-on-chip bus, a converter translates bus protocol requests to this generic memory/register interface.

Back-end: Describes the target platform dependent part of the overall synthesis flow ⑥ (for

more details see Figure 7.27). For the proof-of-concept implementation of the proposed synthesis flow the Xilinx Platform Studio (XPS) [239] and the Xilinx Integrated System Environment (ISE) [238] have been used to implement the back-end flow. It can be subdivided into the following steps:

1. *Platform generation, low-level synthesis, mapping, and place & route* describes the target platform dependent IP configuration, RTL, logic synthesis and place & route. The *platform generation* builds the top-level structural design as synthesizable VHDL. It contains all IP and custom hardware components as synthesizable VHDL or Verilog. The *low-level* synthesis performs RTL and logic synthesis of the entire platform. During *mapping* the logic and memory elements after low-level synthesis are mapped to the components (Lookup-Tables (LUTs), Flip-Flops (FFs), Block-RAMs (BRAMs), and special Hardware Multiplier (MULs)) of the specific target FPGA. Finally, the *place & route* process determines the geometrical location of the LUTs, FFs, BRAMs, and MULs, and determines the routes of the connections between these elements.
2. *Cross compilation of the software part of the design* performs a cross-compilation of the extracted Software Tasks, the peripheral drivers, and the RMI drivers, to an ELF (Extensible Linking Format) file for each target CPU. Depending on the memory subsystem configuration (CPU with data and instructions only in local memory or shared memory) different boot loader instructions are added for staring the software tasks from memory. Finally different linker scripts are applied to each ELF file for mapping data and instructions into proper memory locations. The result of this step is a target platform configuration dependent memory image for each software task.
3. *Bit stream initialization and downloading to the hardware platform* describes the last step for programming the FPGA. The bit stream for programming the configuration SRAM of the FPGA contains the configuration for each LUT, FF, BRAM, MUL and routing resource obtained from hardware synthesis. Local instruction and data memory (implemented in BRAMs) is also initialized with the proper memory images for booting and executing the software.

The Front- and the Back-end including the different sub-phases will be explained in more detail during the following sections.

7.3 Parsing and Intermediate Representation

For parsing the OSSS Application Layer design we are using the EDG C++ front end [217]. The front end does syntax and analysis, including complete error checking. The front end translates source programs into a high-level, tree-structured, in-memory intermediate language. The intermediate language preserves a great deal of source information (e.g., line numbers, column numbers, original types, original names), which is helpful in source analysis and transformation applications.

Implicit and overloaded operations in the source program are made explicit in the intermediate language, but constructs are not otherwise added, removed, or reordered. The intermediate language is not machine dependent (e.g., it does not specify registers or dictate the layout of stack frames).

The front end also includes:

- a C-generating back end, which can be used to generate C code for C++ programs,
- a C++-generating back end, which is useful for source-to-source transformation applications,
- a pre-linker, which handles automatic template instantiation,
- utilities to write the intermediate language to a file, read it back in, and display it in human-readable form,

- and a name demangler.

Both the internal *Fossy* data structure and the intermediate format of the front end are based on the same ISO/IEC C++ standard [14]. The EDG front-end parser is used and augmented with a thin adaptation layer converting the front end specific intermediate format to XML, which is the input for *Fossy* and the Software Extraction (SMOG).

In order to simplify and to accelerate the high-level synthesis a special OSSS synthesis library is used. This library includes reduced header files for all OSSS modeling elements. The same reduced header files are provided for the SystemC library. Since *Fossy* and the Software Extraction (SMOG) do not rely on the analysis of all method bodies most of them can be omitted. This results in a significant speed up during the parsing step.

The central data structure *Fossy* and SMOG are operating on is the OSSS AST (**A**bstract **S**yntax **T**ree). Note that the AST also contains non-synthesizable constructs like `new`, `delete`, pointers, floating points, etc. The reason for this is:

- We have a C++ front-end which more or less directly provides this kind of information.
- The *Software Extraction* operates on the same AST. Since there are no restrictions on software tasks, every C++ construct may occur.
- There are some "semi"-synthesizable constructs like `new` (which may be used to instantiate a module) have to be forwarded to *Fossy*'s elaborator and therefore have to be represented internally.

7.4 Target Platform Representation

As already stated in Section 2.2, basic architecture building blocks of embedded hardware/software systems can be classified in the following categories: software processors, memories and (user-defined) hardware blocks. These components can be interconnected by a communication network, like a shared bus or a high-speed point-to-point connection.

Figure 7.3 shows the building blocks of the Virtual Target Architecture class library. These components are supported in the synthesis flow and can be used to assemble the Virtual Target Architecture. For the sake of simplicity, we only provide a few architecture building blocks instead of a huge library covering all IP components provided by Xilinx.

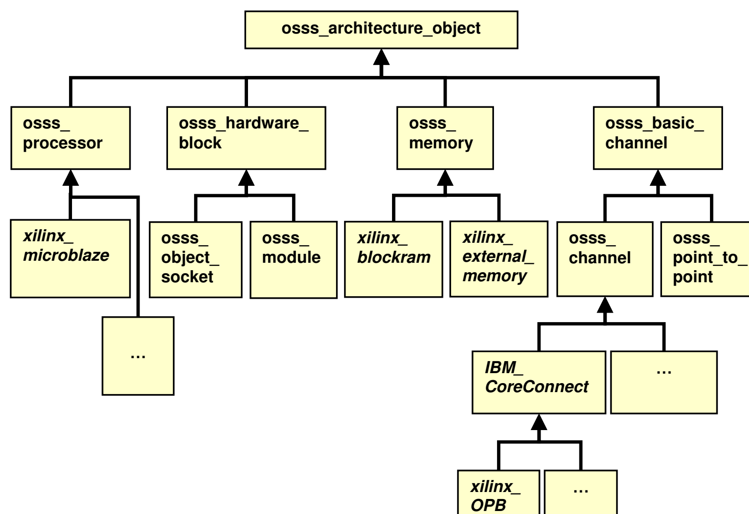


Figure 7.3: Architecture class library [44]

For the proof-of-concept implementation and evaluation of the presented synthesis flow, only a limited set of target platforms can be represented. In general, the choice of the target platform is a crucial issue in the design process embedded hardware/software systems. It is

driven by multiple objectives for finding the "best" or "most suitable" target platform for a given application. The terms "best" or "most suitable" need to be defined with respect to given design and market constraints (e.g. speed, costs, power consumption or flexibility).

In the context of this work the choice of supported target platforms has been mainly driven by accessibility and maturity of the platform configuration and synthesis process to be integrated as back-end into the proposed flow. Combined with the requirements from Chapter 2 the number of possible target platforms has been further narrowed. At the same time, the set of suitable target platforms should not be too restricted and adapted only for special hardware/software designs, but should be generally acceptable for a big class of embedded hardware/software designs.

The traditional way to prototype a system on chip was to assemble a PCB from the basic system components which consist of processing elements ("off-the-self" Micro-Processors, Micro-Controllers or Digital Signal Processors (DSPs)), Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). In these prototyping environments ASICs and FPGAs were used to implement the custom hardware components of the system. Due to the availability of platform FPGAs which are big enough to implements an entire system on chip, it became possible to use special prototyping and development boards carrying one FPGA together with generic peripherals (like interface components and memories). Platform FPGAs are available with either "hard" or "soft" processor cores plus different integrated peripherals and a lot of soft IP components. The two market-leading companies for platform FPGAs are Altera and Xilinx.

Beside platform FPGAs from Altera and Xilinx there are also other FPGAs on the market that basically differ in the internal structure and technology of the FPGA. For the implementation of complex embedded hardware/software systems Altera and Xilinx offer the biggest portfolio of FPGA technology optimized IP components, including high-performance soft-core processors, and highly productive IP configuration and synthesis tools.

In this work target platform support is limited to Xilinx platform FPGAs and prototyping boards as described in Appendix E. Figure 7.4 shows an example of a virtual target architecture composed of different Xilinx specific specializations of generic `osss_architecture_objects`. It includes a single Xilinx MicroBlaze processor block connected to a `xilinx_OPB` as a bus master. A Xilinx Block-RAM and an `osss_object_socket` are connected to the OPB as slave components as well. Two `sc_modules` are connected to the `osss_object_socket` using a simple point-to-point connection (`osss_point_to_point`).

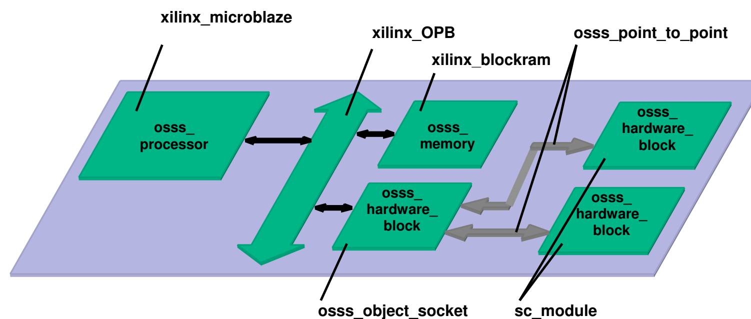


Figure 7.4: Example of a virtual target architecture [44]

Each of the supported architecture building blocks shown in Figure 7.3 will be explained in more detail during the following subsections.

7.4.1 Software Processor Block

The software processor architecture building block is presented using the Xilinx MicroBlaze, which is used in the experiments of this work. The `xilinx_microblaze` is composed of several other IP cores. The heart of this software processor block is the Xilinx MicroBlaze soft processor core, which can configured by several options.

The Xilinx MicroBlaze embedded soft-core processor [103] is a reduced instruction set computer (RISC) optimized for the implementation in Xilinx field programmable gate arrays (FPGAs). Figure 7.5 shows a diagram of the MicroBlaze core. The backbone of the architecture is a single-issue, 3-stage pipeline with 32 general-purpose registers, an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupt. This basic design can then be configured with more advanced features such as barrel shifter, floating-point unit (FPU), caches, exception handling, debug logic, and others.

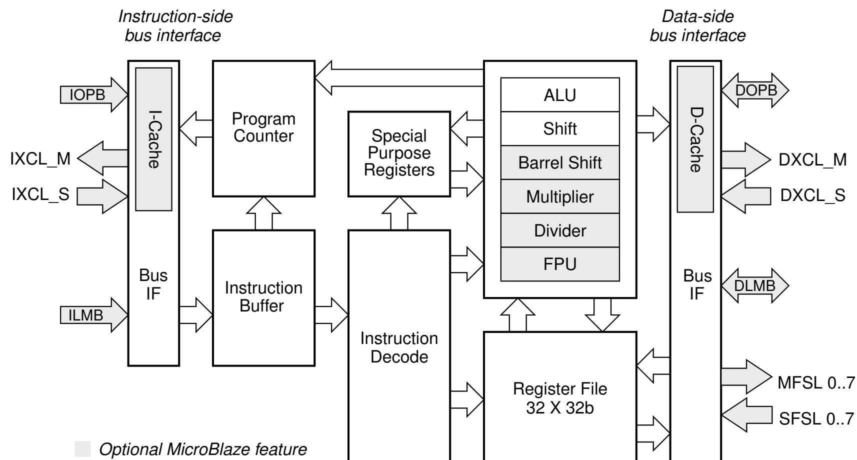


Figure 7.5: MicroBlaze core block diagram [103]

Figure 7.6 shows the internal organization of the `xilinx_microblaze` architecture building block with internal memory. Since the MicroBlaze has a 32 bit RISC Harvard-Architecture it needs separate instruction and data memories. The `xilinx_microblaze` architecture block shown in Figure 7.6 encapsulates both the memories in separate parts of a Block RAM (BRAM). The BRAM is capable of storing a maximum of 64 Kbyte of data and 64 Kbyte of instructions. For the communication with user defined hardware blocks and its peripherals the MicroBalze processor core is connected to an OPB (On-Chip Peripheral Bus) using the DOPB port. The MicroBlaze together with its local data and instruction memory constitute the core components of the `xilinx_microblaze` block.

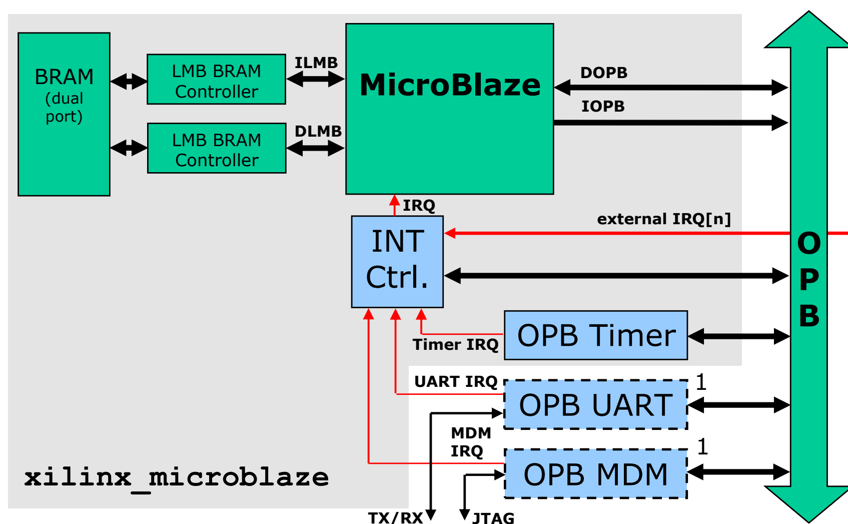


Figure 7.6: Organization of the `xilinx_microbalze` architecture building block (internal memory, no OPB burst) [44]

An interrupt controller can be integrated if either interrupts from user defined hardware blocks or other peripherals (OPB Timer or OPB UART) are necessary. An OPB Timer can be added if the software running on the MicroBlaze makes use of special functions dealing with explicit timing behavior (e.g. software waiting for a certain amount of time). The OPB UART and OPB MDM modules are supported to enable a convenient software debugging environment when running applications on the prototyping board. Considering a system with multiple `xilinx_microblaze` blocks, only one of them can be equipped with an UART for debugging. This restriction results from the limited physical resources on the prototyping board (e.g. the Xilinx ML401 is equipped with only one single SUB-D9 connector usable for serial communication). Since a single OPB MDM (Microprocessor Debug Module) is capable of handling more than one MicroBlaze, only one of them will be shared among multiple `xilinx_microblaze` blocks connected to the same OPB.

The `xilinx_external_memory` IP component can be used to store data and instructions when the size of the local BRAM is not big enough to store the entire program (i.e. the Software Task). Figure 7.7 shows the internal organization of the `xilinx_microblaze` architecture building block with external memory. This version of the `xilinx_microblaze` has to be used if the software application (either data or instructions) does not fit into the local block RAMs. In this case, the data and instruction memory of the MicroBlaze is part of an external memory. It is accessed by the DOPB (for data) and IOPB (for instruction) port, both connected to `OPB1`. For accessing the external memory, an appropriate memory controller has to be instantiated. To speed up the access to the external memory local data (D-Cache) and instruction (I-Cache) caches (both implemented in block RAMs) can be instantiated (not shown in the figure).

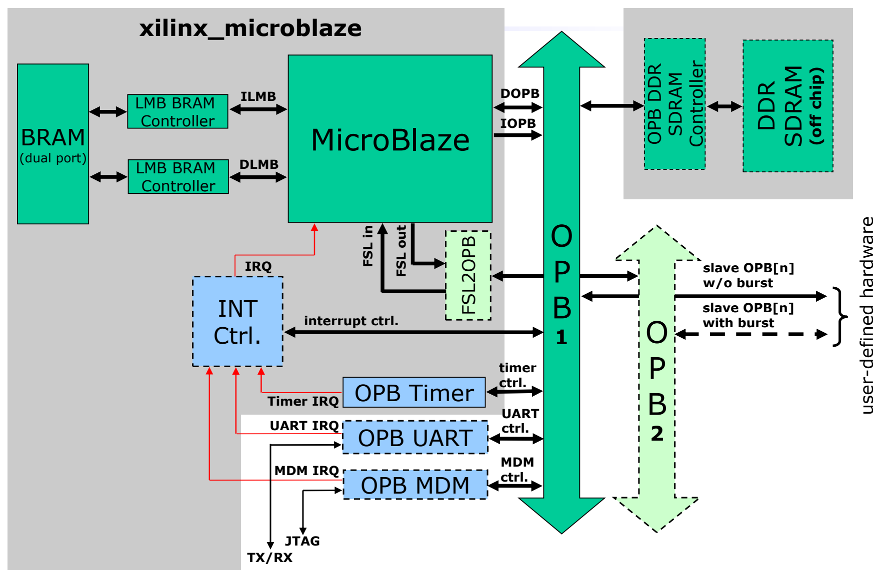


Figure 7.7: Organization of the `xilinx_microblaze` architecture building block (external memory, with OPB burst) [44]

Additionally, `OPB1` is used for configuration and access of the interrupt controller, the timer, the UART and the MDM. When a user-defined hardware block (i.e. a Shared Object) is attached to `OPB1` the RMI protocol operates in single cycle/beat mode only, because the MicroBlaze is not capable of initiating burst transfers on its DOPB or IOPB port. The same applies for the internal memory configuration, show in Figure 7.6 before.

To speed-up data transfer, `OPB2` can optionally be connected to an FSL-to-OPB bridge for enabling the initiation of burst transfers to Shared Objects. Besides the burst capability it can be used to decouple the communication with an external data and instruction memory connected to `OPB1` from the communication with Shared Objects (using `OPB2` via the FSL-to-OPB bridge). This enables a more accurate estimation of basic-block execution times, given that the `OPB1` and the external memory is used exclusively by a single processor.

The MicroBlaze contains eight input and eight output Fast Simplex Link (FSL) interfaces

(see MFSL 0..7 and SFSL 0..7 in Figure 7.5). For the connection to the FSL-to-OPB bridge one input and one output FSL interface of the MicroBlaze is used. The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces on the MicroBlaze are 32 bits wide. The same FSL channel can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information. The FSL is a special I/O mechanism of the MicroBlaze that is directly mapped into its register file. The put and get instructions of MicroBlaze can be used to transfer the contents of a MicroBlaze register onto the FSL and vice-versa.

To conclude, Table 7.1 shows the configurable features of the supported MicroBlaze system.

7.4.2 Hardware Block (osss_hardware_block)

The osss_hardware_block architecture elements represent the user defined hardware at Application Layer mapped to the Virtual Target Architecture Layer. The Application Layer provides two basic building blocks for user defined hardware: Hardware Modules (osss_module) and Shared Objects (osss_shared<...>). The basic difference between these application building blocks is their behavior in terms of activity. Since the module can contain any number of processes, it is able to initiate action on its own. That is why we consider modules with processes as active. Unlike the module, the Shared Object is passive. It provides a method interface, which can be accessed by any hardware module or software task through a port to interface binding. Thus, a Shared Object is activated when the process of a hardware module or software task calls a method on the Shared Object and it loses its activity when the method call has been finished.

In order to reflect the different behavior of a hardware module and a Shared Object, as described above, the following representations on the Virtual Target Architecture Layer are provided:

Hardware Module All user defined hardware blocks described by sc_module on the Application Layer can be realized as osss_module on the Virtual Target Architecture Layer. An sc_module can be easily replaced by an osss_module through replacing the base class, replacing SC_THREAD processes by SC_CHTREAD processes and using the pre-defined clock and reset ports.

Shared Object All Shared Objects on the Application Layer have to be wrapped by an OSSS Object Socket (i.e. osss_object_socket<osss_shared<...> >) when mapped to the Virtual Target Architecture Layer. Thus, OSSS Object Sockets are architectural representatives of the passive Shared Objects at Application Layer.

7.4.3 Memory (osss_memory)

The osss_memory architecture building block can be further subdivided into on-chip (xilinx_blockram) [94] and off-chip (xilinx_external_memory) memory (see Figure 7.8). Both types of memory can be used for hardware/software, software/software or hardware/hardware communication. Both memories are attached as slaves to the OPB. A memory controller is used for accessing the two different memories [125, 106].

The xilinx_blockram module can be instantiated multiple times, depending on the amount of on-chip memory available in the used FPGA. Each xilinx_blockram module has a configurable size from 512 Byte up to 128 kByte.

The xilinx_external_memory module can only be instantiated once, since it uses an off-chip memory module which is mounted on the prototyping board and which is only available once. The xilinx_blockram module has a fixed size depending on the prototyping board (e.g. the Xilinx ML401 prototyping board has a 64 MB of DDR SDRAM).

7.4.4 Communication Network (osss_basic_channel)

For the interconnection of the architecture building blocks, a special kind of channel called OSSS-Channel is used. This channel serves two purposes:

Table 7.1: Configurable features of the MicroBlaze soft core processor

MicroBlaze configurable feature	**Description**
On-chip Peripheral Bus (OPB) data side interface	Used for communication with user defined hardware blocks and/or controller of external memory. For more details of the OPB see Section E.3.6.
On-chip Peripheral Bus (OPB) instruction side interface	Used for communication with controller of external memory. For more details of the OPB see Section E.3.6.
Local Memory Bus (LMB) data side interface	Used to access data (i.e. static data, stack and heap) mapped to the local memory (implemented as BRAM). For more information of the LBM see Section E.3.1.
Local Memory Bus (LMB) instruction side interface	Yes. Used to access instructions (i.e. the software application running on this processor) mapped to the local memory (implemented as BRAM). For more information of the LBM see Section E.3.1.
Fast Simplex Link (FSL) interfaces	Cannot be used explicitly, but is the interface to connect with the FSL2OPB bridge to support burst transfer via OPB.
Instruction and data cache	Supported in combination with external memory configuration (see Figure 7.7).
Hardware barrel shifter	Special hardware (instruction) support to speed-up arbitrary bit-shifting (e.g. used during serialization).
Hardware divider	Special hardware (instruction) support to speed-up integer divisions.
Floating point unit (FPU)	Special hardware (instructions) to speed-up calculation with floating point numbers (`float` and `double`).
Timer	Supports measurement of software execution times and run-time assertions based on RET annotations. The timer (see Section E.3.3) is configured via OPB and can optionally trigger an interrupt (e.g. when the counter hits a certain value) via an interrupt controller.
Universal Asynchronous Receiver Transmitter (UART)	Supports writing into a serial console or debug console and supports `std::cout` streaming. The UART (see Section E.3.4) can also be used to send and receive arbitrary data. It can be connected to the interrupt controller (e.g. to notify the processor when the UART send or receive buffer is getting empty or full).
Interrupt Controller	Supports to connect arbitrary external interrupt sources to the processors interrupt pin. The interrupt controller (see Section E.3.2) is configured and accessed via the OPB.
Microprocessor Debug Module (MDM)	Supports connection with a software debugger (see Section E.3.5).

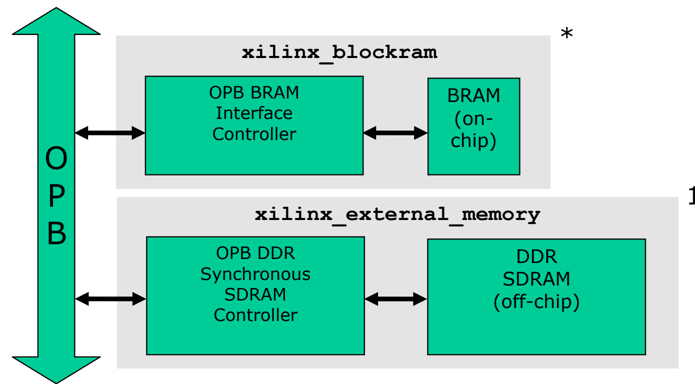


Figure 7.8: Organization of the `osss_memory` blocks (on-chip `xilinx_blockram` & off-chip `xilinx_external_memory`) [44]

1. Encapsulation of a user-defined communication protocol. That means the channel has specific communication interfaces for each connected architecture block and the communication network connecting these interfaces.

2. Representation of architecture building blocks where the communication links of the application layer can be mapped on.

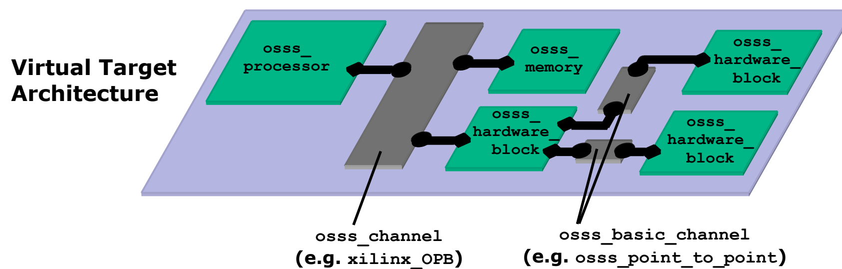


Figure 7.9: Channels connecting the architecture building blocks [44]

Figure 7.9 shows the channels as architecture elements, which interconnect the architecture building blocks. Details of the communication protocol and the physical wires forming the connection network are hidden by them.

We can distinguish between two kinds of channels:

- The `osss_basic_channel` can be used to describe a single master/multiple slave interconnection where each communication is modelled as a broadcast from the master to all the connected slaves. A simple point-to-point connection (`osss_point_to_point`) can be considered as a special case of this topology.

- The `osss_channel` can be used to describe a bus topology with multiple master and multiple slave components. For the support of multiple masters, an arbitration unit with exchangeable scheduling policies will be supplied. For further information refer to [77, 78].

7.4.4.1 Point-To-Point Communication (`osss_point_to_point`)

The `osss_point_to_point` channel is a simple point-to-point connection between user defined hardware blocks and is completely customizable by the designer. We will provide a generic `osss_point_to_point` channel in the virtual target architecture class library with a fixed communication protocol and user definable communication data widths. Nevertheless, it is possible to write completely user defined point-to-point connection and to integrate them into the architecture class library.

7.4.4.2 Bus Communication (osss_channel)

On the one hand, a point-to-point communication architecture achieves a great performance but on the other hand for huge designs many physical wires are necessary and this would lead to huge area consumption. Bus communication tries to overcome this disadvantage. A bus provides a physical channel, which is shared among its connected components. Furthermore, a bus provides a scheduling mechanism for shared medium accesses. This bus scheduling is usually done by an arbitration unit, called bus-arbiter.

In this work, a bus based communication will be used to cross the hardware/software boundary. Because a CPU is usually connected to a bus in order to communicate with its environment, we assume at least a single bus connected to a single CPU in each hardware/software system.

Typical bus systems used in SoC design are ARM AMBA (Advanced Microcontroller Bus Architecture), IBM CoreConnect, Altera Avalon and Opencores Wishbone. For an overview of some important SoC bus standards see [145, 84]. Since the chosen MicroBlaze has an IBM CoreConnect On-Chip Peripheral Bus (OPB) interface we will concentrate on this specific bus system (see Section E.3.6).

7.5 Platform Synthesis

The synthesis flow starts with an OSSS hardware/software design which has been mapped from the *Application* to the *Architecture Layer*. Therefore all communication links on the *Application Layer* have been refined by OSSS-Channels, all software tasks are mapped on certain processors, Shared Objects are wrapped by `osss_object_socket`s and all `sc_module`s have become `osss_module`s. That means that all design building blocks are either architecture building blocks from the architecture class library or wrapped by them.

Figure 7.10 visualizes the architectural context extraction and hardware/software architecture synthesis for the chosen prototyping board Xilinx ML401 (see Section E.2).

In the first step of the synthesis flow the OSSS HW/SW design is elaborated and architectural context information used for further synthesis steps is extracted. This is done by the architecture synthesis part of the OSSS synthesis library that utilizes the SystemC kernel to extract the design structure. The necessary information is extracted at the end of elaboration phase of SystemC. After extracting and analyzing the structure of the design the following intermediate data is generated:

1. Architectural context information stored in an XML format that can be used by the *Fossy* high-level-synthesis tool and the OSSS software library generator
2. Microprocessor Hardware Specification (MHS) used by the Xilinx Embedded Development Kit (EDK)
3. Microprocessor Software Specification (MSS) used by the Xilinx EDK
4. User constraint file (UCF) used by the Xilinx Integrated Software Environment (ISE)

The MHS and the MSS are both Xilinx EDK proprietary formats. The UCF is Xilinx ISE proprietary and the architectural context information XML file is *Fossy* proprietary.

The goal of the architecture extraction is the generation of a 3rd party tool specific architecture description from the VTA Model. Furthermore, the mapping relations between the Application and the VTA model are analyzed and forwarded to *Fossy*. The *Fossy* input is the Application Model only. All mapping and refinement information from the VTA Model that are necessary for the HW synthesis of *Fossy* are stored in a so-called ACI (**A**rchitectural **C**ontext **I**nformation) file.

The architecture extraction process uses the *SystemC*/OSSS *structure reflexion* capabilities that are available before, during and at the end of the SystemC elaboration phase. In a first step a *system-level design rule check* is applied. It checks whether all architecture building blocks are connected properly and all mapping relations are valid. When no design rule is violated, the process continues with a partitioning step. Otherwise, an error message reporting the violation is generated and the synthesis process quits.

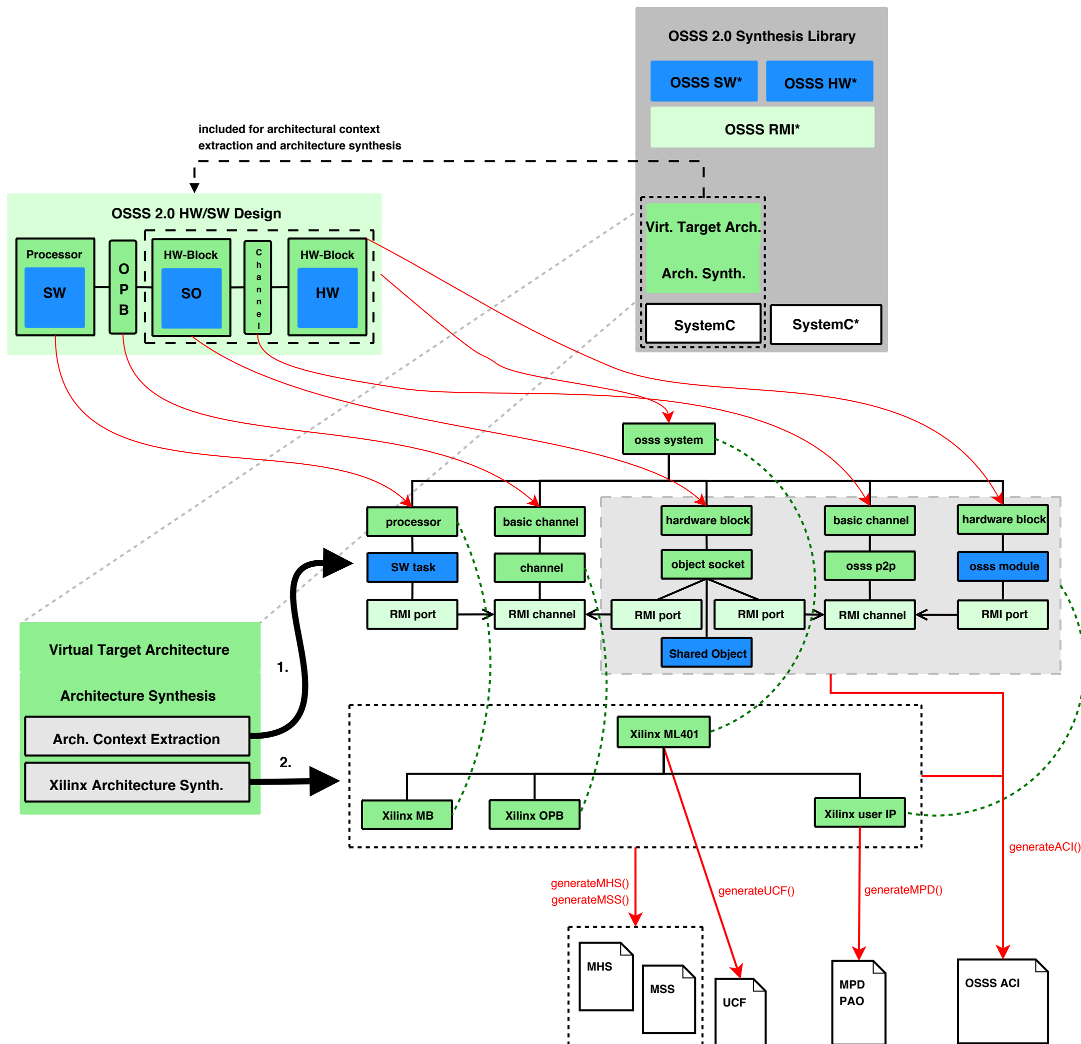


Figure 7.10: Visualization of the architectural context extraction [44]

During *design partitioning* the system is split up into software, hardware, and intellectual-property (IP) partitions. This step is necessary since each partition is treated differently during the subsequent flow.

Software Partition: The software partition consists of a processor together with its peripherals like dedicated data & instruction memory, interrupt controller and timers. Since most processors and its peripherals are connected through a dedicated bus, we also include this bus into the software partition. Finally yet importantly, all software tasks from the Application Model are part of the processor's software partition they are mapped onto. This information is propagated to the *Software Extraction* that extracts the software tasks and all classes it depends on and customizes the OSSS embedded software library (OSSS ESWLib) used for RMI communication with Shared Objects.

Hardware Partition: A hardware partition consists of the maximum set of `osss_modules` and Shared Objects that are connected either by `sc_signals` or synthesizable OSSS-Channels (until now we only support point-to-point channels with user-defined bit widths).

IP Partition: An IP partition consists of a single architecture IP component. E.g. an on-chip bus that should not be generated by *Fossy*, but used as a pre-existing configurable RTL-IP core.

During *Communication Analysis* the mapping from the Application Model's communication

links to the VTA Model's Channels is examined. Both the partition and the communication analysis results are written to the ACI file that is used by *Fossy* during hardware synthesis.

The last step is the generation of architecture definition files for a specific 3rd party tool. Until now, only a Xilinx Platform Studio backend is available. However, other vendor specific backends can be added. The *Xilinx EDK Generator* uses the information collected in the analysis steps above to write an MSS (Microprocessor Software Specification) and MHS (Microprocessor Hardware Specification) file. The first one describes the software properties of each software partition and the second one defines the architecture that is composed out of software, hardware and IP partitions. Each hardware partition is considered as a user-defined IP component whose skeleton is generated in the last step of the architecture extraction process. All necessary hardware description files are to be generated by *Fossy* (see "VHDL dummies").

7.5.1 Architectural Context Information

The idea of the architectural context extraction is based on a structure reflection during run-time. This is possible because most of the design structure information is inherent in the simulation context which is built and managed by the SystemC kernel. During the elaboration phase which is part of the execution of a SystemC application the module hierarchy is built up. SystemC provides a convenient programming interface to access this module hierarchy.

The `end_of_elaboration()` method is part of the standard implementation of the classes `sc_module`, `sc_port`, `sc_export` and `sc_prim_channel`. By default these functions do nothing. They need to be overridden in a user-defined subclass of these classes in order to perform structure reflection. At the end of the elaboration phase no further modifications are allowed to the module hierarchy. Therefore, it is reasonable to use this hook to start the structure reflection.

```
1  virtual void end_of_elaboration() {
2    std::vector<sc_object*> tops = sc_get_top_level_objects();
3    recursive_descent(tops);
4  }
5
6  void recursive_descent(std::vector<sc_object*>& obj_vec) {
7    for(unsigned int i=0; i<obj_vec.size(); i++) {
8      if(sc_module* module = dynamic_cast <sc_module*>(obj_vec[i])) {
9        sc_interface* channel_if = dynamic_cast<sc_interface*>(obj_vec[i]);
10       if (channel_if) {
11         //add hierarchical channel to design hierarchy
12       }
13       else {
14         //add module to design hierarchy
15       }
16       std::vector<sc_object*> children = module->get_child_objects();
17       recursive_descent(children);
18     }
19     else if (sc_port_base* port =
20                 dynamic_cast <sc_port_base*>(obj_vec[i])) {
21       //add port to design hierarchy
22     }
23     else if(sc_prim_channel* channel =
24                dynamic_cast<sc_prim_channel*>(obj_vec[i])) {
25       //add primitive channel to design hierarchy
26     }
27     else if (...) {
28       ...
29     }
30   }
31 }
```

Listing 7.1: Example of structure reflection using the SystemC library

Listing 7.1 gives an example for the usage of the `end_of_elaboration()` method and illustrates how the structure of a SystemC can be traversed. The function `recursive_descent(...)` gets a vector of `sc_object` pointers. The `sc_object` is a base class of each structural design element in SystemC. A module or a port of a module is a specialization of that base class.

Thus the dynamic_cast<...>(...) operator can be used during run-time to check whether a sc_object is a module, a port or a primitive channel.

In addition to the module hierarchy information, the port to interface binding can be extracted during run-time. In this context a detailed knowledge about interfaces (sc_interface) is desirable. Since interfaces are not derived from the base class sc_object and therefore do neither belong to the module hierarchy nor to the object hierarchy it is more difficult to extract them. By making use of the typeid operator of the C++ run-time type information (RTTI) it is possible to get the type name of each port.

This structure reflection mechanism has been extended in order to extract the architectural context information in the OSSS synthesis flow.

Figure 6.18 shows the architecture class library which contains the supported architecture building blocks. This library contains general elements that are independent from a certain target platform. These are the osss_architecture_object base class and the osss_processor, osss_hardware_block, osss_memory and the osss_basic_channel. These classes can be considered as base classes that categorize architecture elements and serve for the extension by more target platform specific elements. Up to now the OSSS Synthesis Library only contains platform dependent building blocks which are IP cores from Xilinx. These are the MicroBlaze soft processor core, Block RAM, external Memory and the On-Chip Peripheral Bus (OPB). The concept of the architecture class library allows also supporting other target platforms like for instance from Altera. In this case the library has to be extended by the IP cores provided by Altera.

As one can see from Figure 6.18 the osss_architecture_object is derived from sc_module, which is derived from sc_object. Moreover all OSSS ports are derived from the appropriate SystemC port class sc_port<IF, N> and all interfaces are derived from the sc_interface base class. Hence it becomes possible to use the reflective approach that has been sketched in Listing 7.1.

As shown in Figure 7.10 the structure reflection starts with the architectural context extraction at the end of elaboration phase. During this phase a tree that represents the structural hierarchy of the OSSS HW/SW design is generated. In Figure 7.10 the architecture of a simple hardware/software design is extracted. The top level module describes an OSSS system that defines the system boundary and connections to the “external world” like clock and reset ports. This system contains a processor that executes a software task with an RMI port that is connected to an RMI channel. Additionally, the system contains a hardware block which is an object socket that contains a Shared Object and two RMI ports. One of the RMI ports is connected to the same RMI channel as the processor and the other RMI port is connected to another RMI channel that represents a point-to-point (OSSS p2p) connection. The last architecture element is a hardware block which is an OSSS module that contains an RMI port that is connected to the point-to-point channel as well.

After the structure reflection has been completed, the top-level design (which needs to be derived from osss_system[2]) is partitioned. We define the following three kinds of partitions:

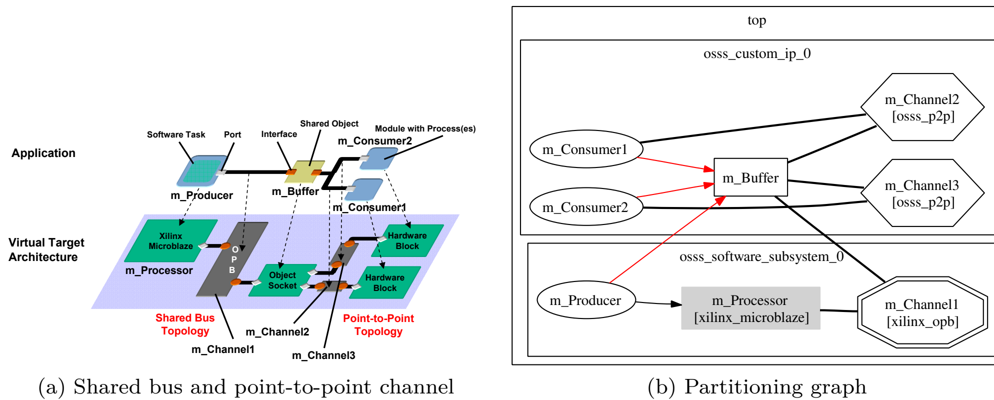
- **Software Subsystem:** A software subsystem contains at least a single processor, its associated software tasks and the bus to which the processor is connected. This partition is considered as an IP block, since both the processor and the bus are taken from an existing IP library of the Xilinx EDK.

- **Custom IP:** A custom IP partition contains any number of hardware modules and Shared Objects. The main restriction on custom IP block is, that they are only allowed to contain point-to-point channels, because these are also custom synthesized. This is an important restriction, because all custom IPs that are identified in this synthesis step are further processed by *Fossy*.

- **Channel IP:** All channels which are not connected to a software processor belong into this kind of partition. A channel IP contains only a single channel. In the current state of the OSSS synthesis flow this can only be a Xilinx OPB.

[2]Because osss_system is the base class to define the SoC boundary (i.e. top-level module for synthesis).

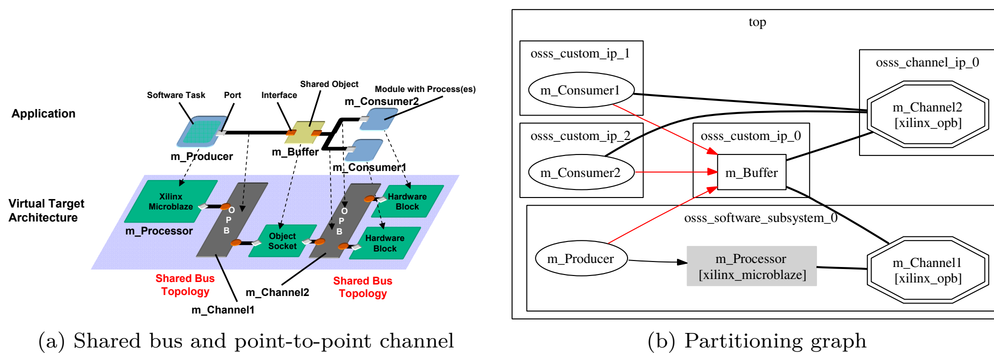
Figure 7.11 shows the result of the system partitioning based on the producer/-consumer example mapping alternative A from Section 6.6.1. The software subsystem `osss_software_subsystem_0` contains the producer software task that is mapped on a processor of type Xilinx MicroBlaze and its associated Xilinx OPB channel. The custom IP `osss_custom_ip_0` contains both consumer hardware modules, the buffer Shared Object and the two point-to-point channels.



(a) Shared bus and point-to-point channel (b) Partitioning graph

Figure 7.11: Partitioning of the producer/consumer example mapping alternative A from Section 6.6.1 [44]

The next Figure 7.12 shows the result of the system partitioning based on another mapping alternative of the producer/consumer example. The software subsystem `osss_software_subsystem_0` stays unchanged compared with Figure 7.11. Since both point-to-point channels from the above example have been replaced by a single Xilinx OPB channel we now have to deal with three instead of a single custom IP. The partitioning results in a custom IP for each consumer hardware module and the buffer Shared Object since all of these components are connected to the second Xilinx OPB channel. The channel itself is encapsulated by the `osss_channel_ip_0` partition.



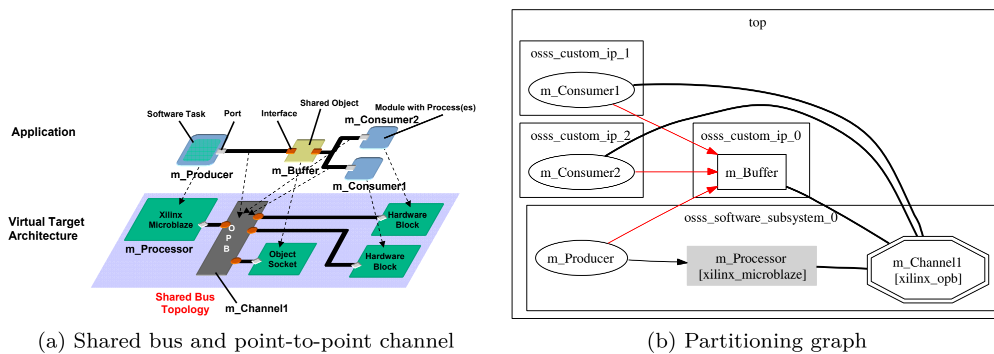
(a) Shared bus and point-to-point channel (b) Partitioning graph

Figure 7.12: Partitioning of the producer/consumer example mapping alternative C [44]

The last Figure 7.13 shows the result of the system partitioning based on the mapping alternative B of the producer/consumer example from Section 6.6.1. Compared to Figure 7.12 the only difference is the missing Xilinx OPB channel `m_Channel2` because all communication links from the *Application Layer* have been mapped on `m_Channel1`.

The step of the architectural context extraction is followed by the target platform specific architecture synthesis. This is done by looking at the most specialized target specific class of each architecture element identified during architecture context extraction.

Considering the example from Figure 7.10 the OSSS system is specialized by the Xilinx ML401 development board. This includes the specific ports from the Xilinx Virtex 4 FPGA to



(a) Shared bus and point-to-point channel (b) Partitioning graph

Figure 7.13: Partitioning of the producer/consumer example mapping alternative B from Section 6.6.1 [44]

the devices mounted on the ML401 board. This includes the input clock, reset, external memory and I/O pins. The processor is specialized by a Xilinx MicroBlaze that is connected to an OPB.

Out of the Xilinx specific components the MHS (Microprocessor Hardware Specification) and the MSS (Microprocessor Software Specification) file for the EDK is generated. The Xilinx ML401 system description is used to generate a target platform dependent UCF (User Constraint File) that contains the physical position of the clock, reset and external memory pins. In order to pack the user-defined IP for using it in the Xilinx EDK an MPD (Microprocessor Peripheral Definition) and a PAO (Peripheral Analyze Order) file are generated.

The OSSS ACI (**A**rchitectural **C**ontext **I**nformation) file is basically generated from the information gained during architectural context extraction and gets enriched by some of the target platform dependent information. In general it contains all the information about the internal structure of the user-defined hardware part of the system that is further processed by the high level synthesis tool *Fossy*. Additionally it contains information about how this user-defined hardware is integrated and connected to the overall system architecture defined by the MHS.

In the following sections the generated files will be described in more detail.

7.5.2 MHS and MSS Generation

The Xilinx EDK builds on top of the Xilinx ISE and can be regarded as an entry for the ISE synthesis flow. The aim of the EDK is the integration of various IP components including embedded processors, DSP blocks, peripherals and communication IPs. The EDK itself includes a library of several IP components including the MicroBlaze embedded soft core processor plus several peripherals which can be interconnected by using the IBM CoreConnect technology. Since the MicroBlaze is a soft processor core it can be fully customized. By using the Xilinx EDK a designer can assemble an architecture consisting of different IP components including user-defined hardware blocks. After the assembly and configuration of the desired architecture the Xilinx ISE is used to synthesize and download the whole design to an FPGA.

In the proposed synthesis flow the Xilinx EDK serves three purposes:

1. It is used to generate the hardware architecture of the hardware/software design. This includes the generation of black-box placeholder components for the integration of the *Fossy* synthesis output into the overall system architecture.

2. Provides the cross-compiler tool chain for the target processor (MicroBlaze, PowerPC)

3. Serves as a front-end to the Xilinx or Synplicity synthesis tools

During this section we will concentrate on the creation of an EDK project by means of generating an MHS and an MSS file.

The MHS file defines the system architecture consisting out of peripherals, and embedded processors. It also defines the connectivity of the system, the address map of each peripheral

in the system, and the configurable options for each peripheral. It is also possible to specify multiple processor instances connected to one or more peripherals through one or more buses and bridges. Since the MHS file is a simple text file it can be created and examined easily by using a text editor.

This simple file format enables the automatic creation of an MHS. The generation of an MHS file is performed by the building blocks of the Virtual Target Architecture. Each class of the architecture class library has a method that appends its configuration to the MHS file. The MHS file as well as the architectural context information is generated during the end of elaboration phase.

The Xilinx EDK Platform Generator tool (Platgen) creates the hardware platform using the MHS file as input. Platgen creates netlist files in various formats such as EDIF and Xilinx proprietary NGC and top level HDL wrappers to allow the addition of user-defined components to the automatically generated hardware platform.

The software platform is defined by the Microprocessor Software Specification (MSS) file. The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is also a simple text file and thus can be treated like the MHS.

The creation of the MSS file is analogue to the MHS file generation. It is performed during the end of elaboration phase of the SystemC kernel. Each class of the architecture class library has a method that appends information about its software driver to the MSS file.

The MHS and the MSS file are inputs to the Xilinx EDK Library Generator tool (Libgen) for customization of drivers, libraries, and interrupt-handlers.

7.5.3 UCF Generation

Details such as I/O pin mappings and timing constraints cannot be expressed in Verilog or VHDL, but are nonetheless important considerations when implementing the design on real hardware (i.e. an FPGA). The UCF file is an ASCII file specifying physical constraints on the logical design.

In the proposed synthesis flow the UCF file is generated from the information contained in the top-level entity of the OSSS design on the *Architecture Layer*. Its main purpose is the mapping of logical ports defined in the top-level design (such as clock, reset or user-defined I/O pins) to the physical port locations of the FPGA. Most of this mapping can be done automatically when the designer specifies the used kind of prototyping board.

The UCF file is generated during the end of elaboration phase of the SystemC kernel. It is used as input to the Xilinx ISE Synthesis tools (more precisely the NGDBuild tool that is used to build up the internal representation for the mapping and the place & route phase).

7.5.4 MPD and PAO Generation

The MPD and the PAO files are both needed to wrap the VHDL files generated by *Fossy*. They are used to integrate user-defined hardware blocks as Xilinx EDK compatible IP cores and to define the further processing of the VHDL files by the back-end RTL synthesis tools.

The Microprocessor Peripheral Definition (MPD) file defines the interface of the peripheral. An MPD file has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters and default values
- Any MPD parameter is overwritten by the equivalent MHS assignment

Thus the MPD file defines the MHS representation of the user-defined hardware generated by *Fossy*.

A Peripheral Analyze Order (PAO) file contains a list of HDL files that are needed for synthesis, and defines the analysis order for compilation. This information is needed by the back-end RTL synthesis tools that are used to process the *Fossy* generated VHDL code. Additionally

it defines the compilation order of the software driver during cross-compilation for the target CPU.

7.5.5 OSSS ACI Generation

In order to perform the hardware/software interface synthesis the following information needs to be provided by the architectural context extraction:

- The design structure on the *Application Layer* of the OSSS design at the end of elaboration phase. It consists of `osss_software_task`s and `osss_module`s containing `osss_port`s bound to Shared Objects.
- The design structure on the OSSS *Virtual Target Architecture Layer*. It consists of different kinds of architecture building blocks and connections to either OSSS-Channels or signals. Together with the design structure on the *Application Layer* model the mapping is retained. This includes the channel bindings, i.e. the mapping of the communication links to OSSS-Channels. This implies the number of clients to a Shared Object.
- Object IDs for each Shared Object plugged into an `osss_object_socket`
- Method IDs for each method of a Shared Object
- Client IDs for each client process performing calls on a Shared Object
- OSSS-Channel transactor specific parameters: E.g. channel bit width of a point-to-point channel and each client's address range for bus transactors. The address range of each slave interface of an `osss_object_socket` is calculated in dependance on to the number of clients hidden by the corresponding bus.

This Architectural Context Information (ACI) serves as input for the high-level synthesis tool *Fossy* (see Section 7.7) and the RMI-Types generator for the customization of the OSSS software library (see Section 7.6).

7.6 Software Synthesis

7.6.1 Introduction

In the following sections, the software synthesis with a focus on hardware/software communication is described. The hardware/software communication is based on the remote method invocation concept presented in Section 6.5.2. We assume the reader is familiar with the OSSS-RMI protocol.

In the proposed methodology, we make the following software related assumptions concerning hardware/software communication:

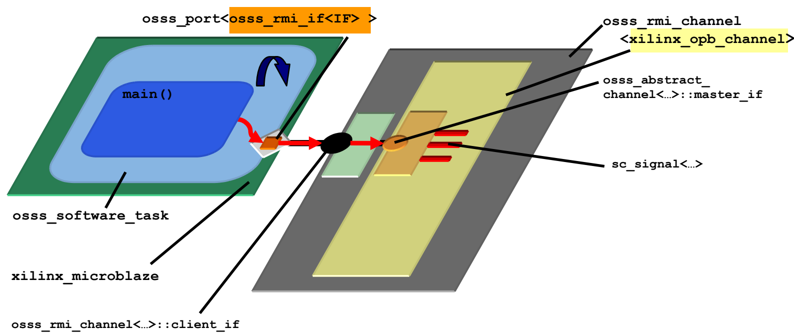
- a processor can only run a single task because on task scheduling is supported in this work. I.e. there is a 1 : 1 mapping relation between a Software Task and a processor.
- a processor is intrinsically tied to at least a single communication network (e.g. a bus)
- a processor is the initiator of a communication (i.e. it is a master in the communication network it is connected to)
- the processor initiates either a sequence of single-beat transfer, a burst transfer or a sequence of burst transfers on its connected communication network
- RMI is implemented as blocking message passing using polling communication. I.e. no interrupts are used for the custom hardware/software communication

In the following sections, we start with an overview of the supported MicroBlaze processor subsystem. This is followed by a definition of the language subset that is allowed to be used for synthesis of Software Tasks. The description of the OSSS software stack containing services for RMI is used to implement the hardware/software communication for Software Tasks. This section closes with a description of the software cross-compiler tool-chain of the synthesis back-end flow.

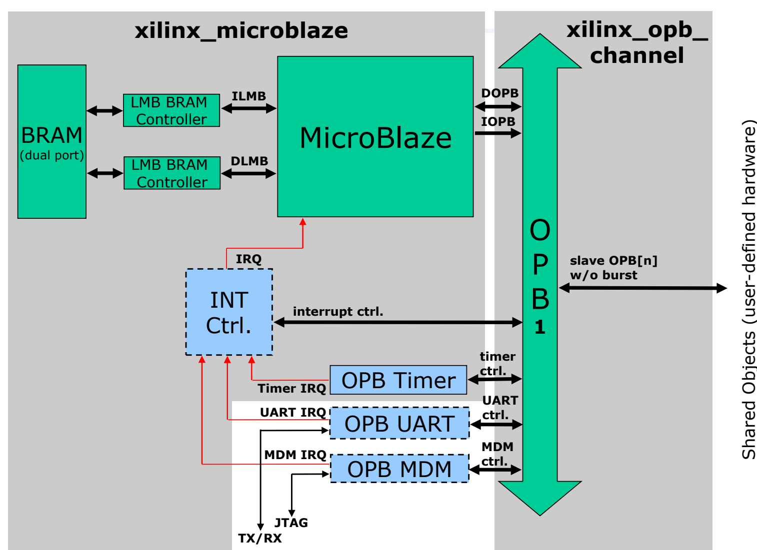
7.6.2 The MicroBlaze Processor Subsystem

Figure 7.14a shows an `osss_software_task` that has been mapped onto a `xilinx_microblaze` processor. The `osss_software_task`'s `main()` routine makes use of an `osss_port<osss_rmi_if<IF> >` for calling a method on a specific interface (`IF`) that is implemented by Shared Object (not shown here). As physical communication medium a `xilinx_opb_channel` is used. The `osss_rmi_channel<...>` wrapper represents the RMI message passing protocol over the underlying physical OPB channel.

After the architectural context extraction phase of the synthesis design flow (as described in Section 7.5) a Xilinx MicroBlaze soft processor core connected to an On-Chip Peripheral Bus (OPB) is generated. Figure 7.14 shows the replacement of the Virtual Target Architecture software subsystem consisting of the `xilinx_microblaze` and the `xilinx_opb_channel` during platform synthesis.



(a) VTA representation of the Software Subsystem



(b) Resulting Target Architecture Software Subsystem (simple)

Figure 7.14: Replacement of Virtual Target Architecture software subsystem during platform synthesis [44]

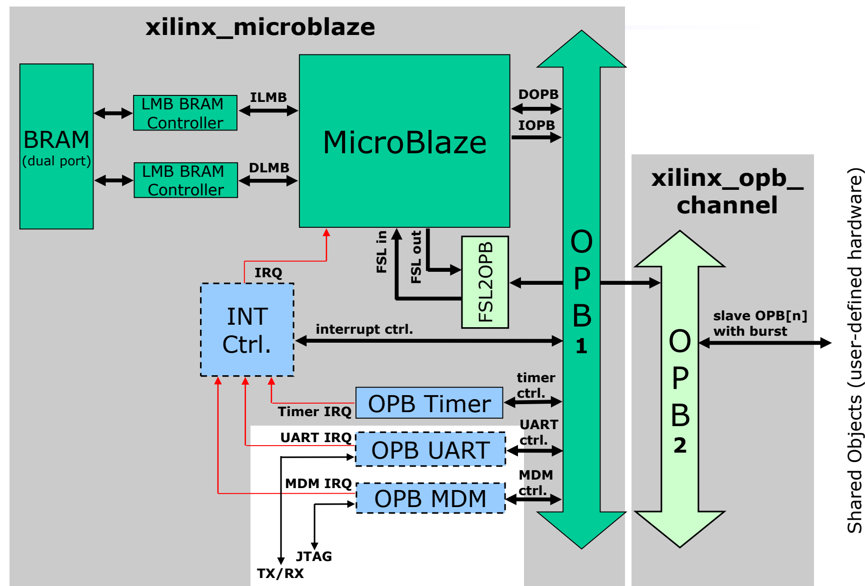


Figure 7.15: Resulting Target Architecture Software Subsystem (dedicated RMI bus) [44]

Figure 7.14b shows the basic/simple MicroBlaze processor subsystem configuration already introduced in Section 7.4.1. The `xilinx_microblaze` component contains a local memory (implemented in a BRAM) for data and instructions (Harvard architecture), a hardware timer and an interrupt controller (only if required by the timer, the UART or the microprocessor debug module (MDM)). The UART and the MDM are not part of the *xilinx_microblaze* component since these blocks are only used for debugging on the target platform. The `xilinx_microblaze` is always connected to at least one On-Chip Peripheral Bus (`xilinx_opb_channel`). The advanced configuration of the MicroBlaze processor subsystem shown in Figure 7.15 uses a dedicated OPB bus for the implementation of the RMI protocol with the connected Shared Objects. The `OPB2` is connected via an FSL-to-OPB bridge which enables burst transactions and and better timing predictability, because only RMI communication is segregated from other bus accesses which are not visible during Virtual Target Architecture Layer model simulation.

For more details on the MicroBlaze processor subsystem refer to Section 7.4.1.

7.6.3 Supported Software Language Subset

As already introduced in Section 6.2, Figure 7.16 shows the relation of OSSS to C++ and SystemC. OSSS is divided into three different subsets: a software description subset, a hardware description subset and an architecture description subset. Each of these subsets defines which modeling elements or language features are allowed in each domain (software, hardware or architecture). There is one intersection between the software and the hardware domain. This intersection includes the language features that may be used in both the hardware and the software domain. This intersection is very important because it defines the hardware/software interface modeling elements. For instance only the data types situated in the hardware/software intersection may be used for hardware/software communication.

The `osss_software_task` is the foundation for the software part of each OSSS design. The red rectangle in Figure 7.16 shows the definition of the OSSS software language subset that can be used inside `osss_software_tasks`.

Each software task has a single `main()` method which is the root function or entry point of each task. The software task's behaviour is implemented inside this `main()` routine. In addition to the `main()` routine the software task contains one or more ports of the type `osss_port<osss_rmi_if<IF> >` that are used for hardware/software communication with Shared Objects. These ports access a user-defined method interface (IF) that is implemented in a Shared Object (in hardware). Hence, this interface only consists of language elements that are

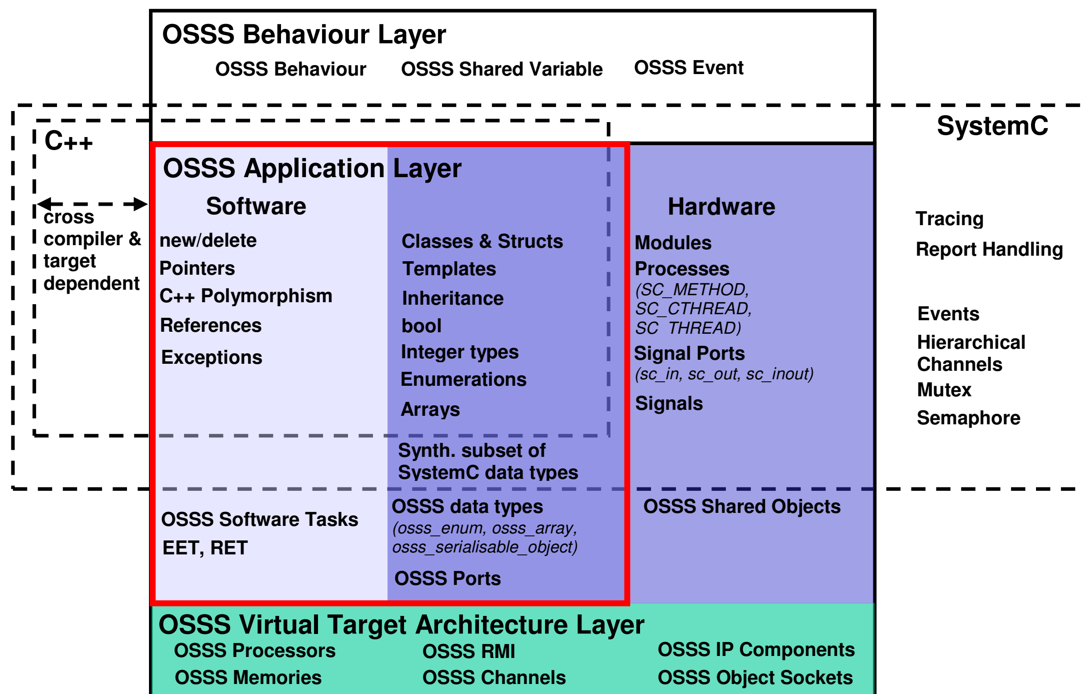


Figure 7.16: The OSSS software language subsets (see red rectangle) [44]

in the OSSS hardware subset. Figure 7.17 gives an overview of the synthesis requirements for software tasks in OSSS.

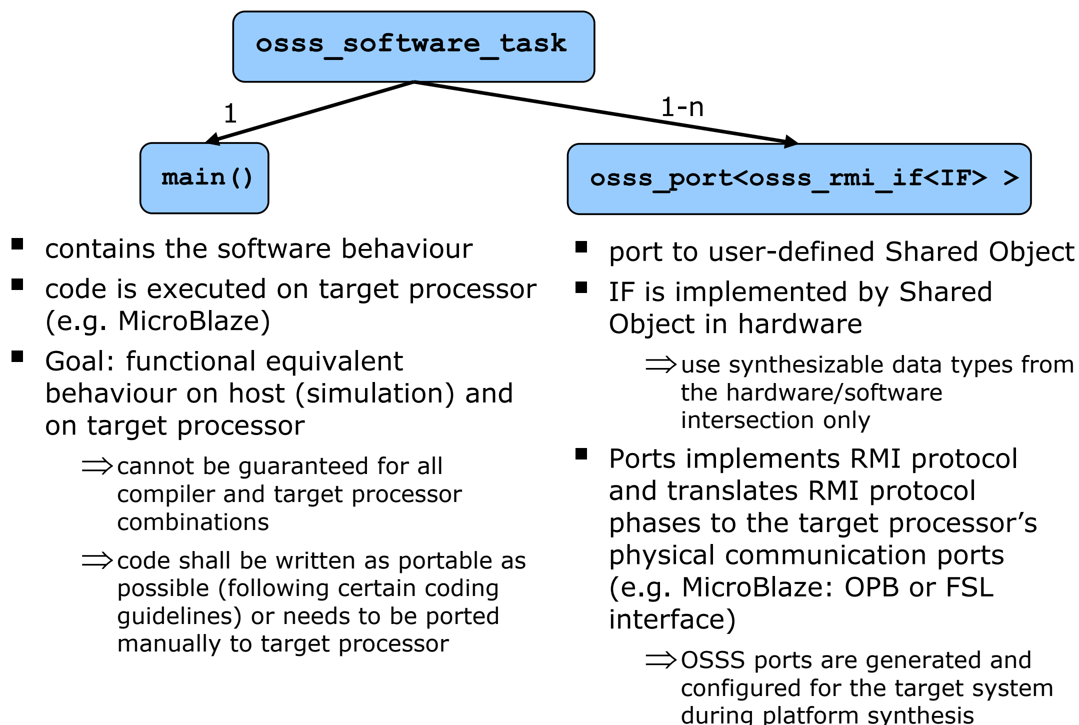


Figure 7.17: Synthesis requirements for Software Tasks in OSSS

Practically, the whole ISO C++ language [14] can be used inside a Software Task's `main()` routine.

An important issue is the functional equivalent behavior of the software code running on the simulation host (Software Task running in Application Layer and Virtual Target Architecture Layer models) and the target processor. For the following reasons C++ is by design not well suited for the design of (safety-critical) embedded systems:

- The semantics of some C++ constructs are not fully specified, leaving room for portability issues the behavior of a program may vary depending on what compiler was used (undefined, unspecified and implementation-defined behavior). E.g. bit fields, but even the size of basic integer types depends on the target processor architecture.
- C++ makes it very easy for a programmer to make mistakes that cannot be diagnosed by a compiler (e.g., the use of the assignment operator "=" instead of the comparison operator "==").
- C++ does not provide any built-in run-time checking, e.g. for arithmetic overflows or array bound errors.
- While being a strongly-typed language, C++ leaves too many holes to circumvent the type system, deliberately or unintentionally.

When using C++ for (safety-critical) embedded systems, great care must be taken to avoid any language constructs and code that can potentially lead to unintended program behavior. C has most of these issues as well, though, and this has not stopped C becoming one of the most widely used languages in safety-critical systems. This has only been possible through the introduction of coding standards that limit language features to a safe subset that can be used without giving rise to concerns.

The most popular coding standard for using C in safety-critical systems is MISRA-C [124]. It specifies a "safe" subset of the C language in the form of 121 required and 20 advisory rules. MISRA-C has also had a great influence on another coding standard for using C++ in safety-critical systems. The "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program" [100], or JSF C++ in short, was kind of revolutionary, as it signaled a move away from Ada as the mandated programming language for avionics software by the US Department of Defense. JSF C++, while taking many rules from MISRA-C, is a bit different in concept from MISRA-C (and the new MISRA-C++) as it also defines coding style and metric guidelines, which the MISRA standards do not have. In total, JSF C++ defines 221 rules. The current trend to move from C to C++ in the development of critical systems has lead to the publication of MISRA-C++ [51] in the summer of 2008. MISRA-C++ takes many rules from MISRA-C and adds many more C++ specific rules, bringing it to 228 rules.

Another issue affecting the portability of C++ code is based on differences in the byte order (also called endianess), the word size, and the memory alignment of complex data types. For hardware/software communication the byte ordering is set to big endian (network byte ordering).

Definition 7.6.3.1 (Endianness):
Endian or endianness refers to the ordering of individually addressable sub-components within the representation of a larger data item as stored in memory or sent on a serial connection. Each sub-component in the representation has a unique degree of significance, like the place value of digits in a decimal number. These sub-components are typically 16-, 32- or 64-bit words, 8-bit bytes, or even bits. □

Figure 7.18 gives a graphical overview of big and little endianess representations in registers and memories.

Definition 7.6.3.2 (Data structure alignment):
When a CPU reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system). Data structure alignment is the way data is arranged and accessed in a memory. It consists of two separate but related issues:

1. ***alignment:*** *means putting the data at a memory offset equal to some multiple of the word size*

[3]taken from `http://commons.wikimedia.org/wiki/File:Endianessmap.svg`

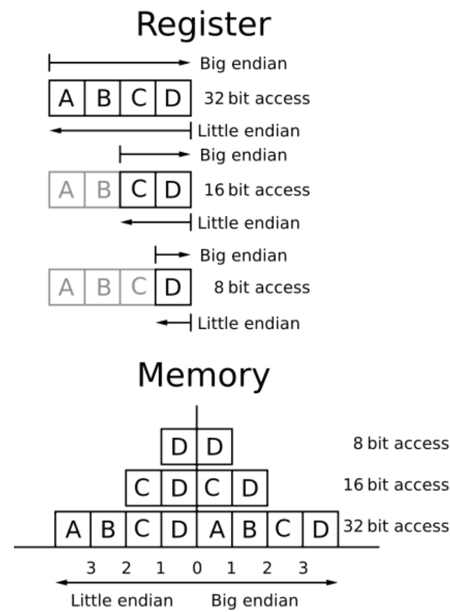


Figure 7.18: Big and Little Endianness representations in registers and memories[3]

2. ***padding:*** *to align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure and the start of the next, which is data structure padding*

□

For the data structure alignment two different strategies have been considered in this work:

Target CPU Alignment: Using this strategy the data alignment of the target CPU is preserved and represented in the register interface of the Shared Object. The advantage of this strategy is that data can be natively represented in hardware. It allows to use `memcopy` for an efficient data transfer from Software Tasks to Shared Objects. The disadvantage is that the Shared Object needs explicit knowledge about the data structure alignment of each client for properly accessing data elements. This client dependent data structure alignment mapping information can be generated during Platform Synthesis and provided to the custom hardware synthesis process using extended Architecture Context Information (ACI).

Packed: This strategy serializes all data structures into a seamless stream. This packing results in the smallest possible representation in memory, that is also the default representation of complex data types in custom hardware. The advantage is that the packed stream represents a normalized data representation that can natively accessed by custom hardware. The main disadvantages are the computational overhead to generate the seamless data stream in software, and the inefficient access when word boundaries are wrapped at memory line boundaries.

For the software subset of OSSS there are no further restrictions and even some non-portable language features of C++ are included. Concerning the OSSS design methodology the software designer has to take care about the portability of software code regarding the targeted execution platform.

We expect that the functional behavior of the software part developed and executed on the simulation host machine is the same after cross compiling and running it on the target processor. In order to make things a little bit more manageable we assume that the compiler front-end used to compile code for the host machine is the same as in the cross compiler. Currently the only supported software compiler front-end is the GNU Compiler Collection for C/C++ (`gcc` respectively `g++`) [222], versions >= 3.4.4. The cross compiler used for the MicroBlaze processor is the `mb-gcc` port that is based on a `gcc 3.4.1` [221]. If possible, the same compiler front-end

for compiling the Application Layer and Virtual Target Architecture Layer simulation models on the host and for the software task cross-compilation shall be used.

7.6.4 The OSSS Software Library & RMI protocol stack

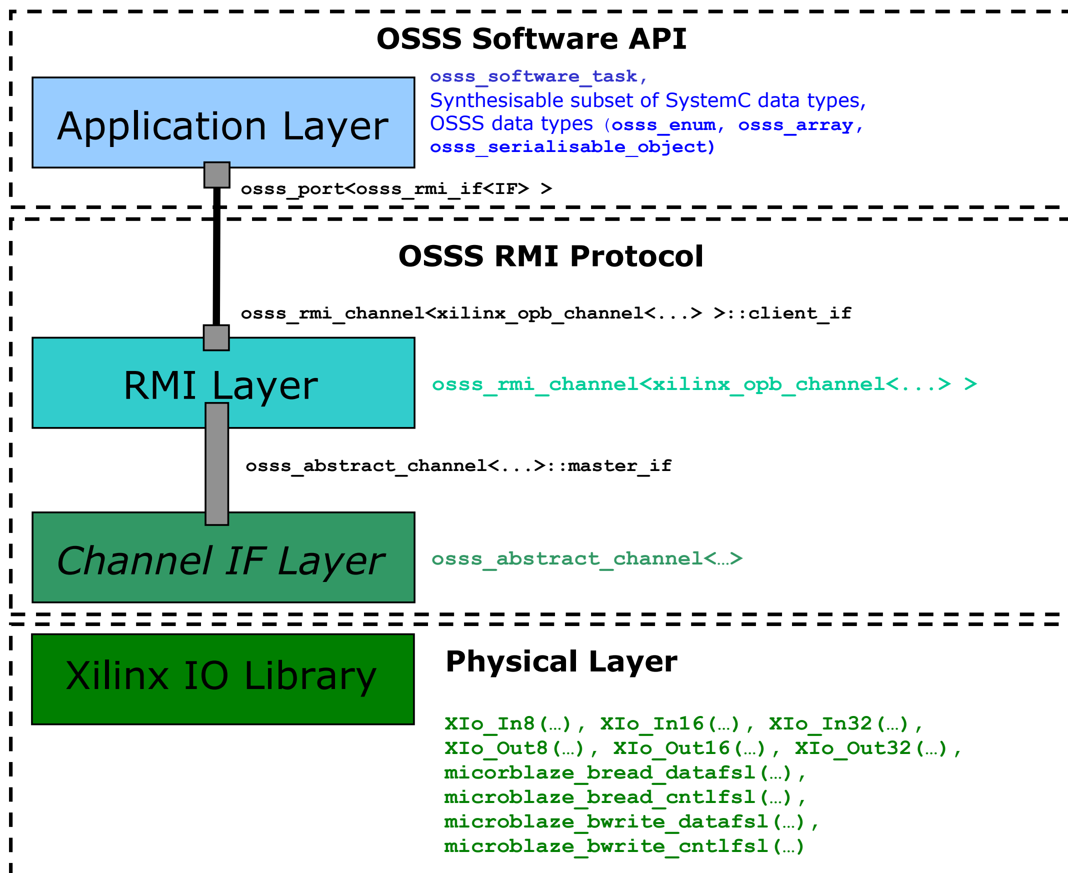


Figure 7.19: The OSSS software library used on the Xilinx MicroBlaze processor [44]

The OSSS software library provides features and functionalities for the software part of an OSSS design that are essential when running it on the target processor. It is divided into the Software API and the RMI protocol stack, see Figure 7.19:

1. The OSSS Software API (software related parts of the *Application Layer*) provides:
    - The same interface to the software designer during simulation and synthesis (i.e. the application designer does not need to change any code of the software part of the system concerning:
        (a) the interaction with the `osss_software_task`,
        (b) the communication with the hardware through `osss_port<...>`
    - Software execution environment (`osss_software_task`)
    - Timing annotations (EETs) are neglected, timing constraints (RETs) can be checked using hardware timer facilities
    - Data types for hardware/software communication (synthesizable subset of SystemC data types and OSSS data types)
    - Interface to the OSSS RMI protocol stack (`osss_port<osss_rmi_if<...> >`)
2. OSSS RMI protocol stack performing the hardware/software communication protocol

The OSSS Software API has been designed to be as target processor independent as possible (see discussion in Section 7.6.3). Even most parts of the underlying OSSS RMI protocol stack are target processor independent. Only the bottommost I/O layer of the OSSS software library is target processor dependent.

Finally, the OSSS software library and RMI protocol stack is cross-compiled for the specific target processor (i.e. the Xilinx MicroBlaze processor) together with the user-code from the Software Task's `main()` routine. Depending on the memory configuration (only local, local + external memory) of the MicroBlaze different linker scripts and boot-loaders are used to generate the memory image.

The following subsections explain the different layers of the OSSS software library & RMI protocol stack.

7.6.4.1 Application Layer

The purpose of the OSSS Software Application Layer API is to provide an environment that with same interface as in the OSSS simulation library. This way software code written for simulation on the simulation host can be cross-compiled to a specific target processor and retains its functionality (with respect to the restrictions discussed in Section 7.6.3).

The *Application Layer* of the OSSS *Software Library* contains the `osss_software_task` that acts as the top-level execution environment for software. Since only a single Software Task can be mapped on a processor, the `osss_software_task` is used as a wrapper for the native `int main()` entry of a C/C++ program.

The software timing during simulation as defined by EET (Estimated Execution Time) blocks are eliminated in the OSSS software library. When running the cross-compiled code on the target CPU itself the timing behavior is inherently defined by the execution order and the interpretation of the instructions on the target CPU.

Listing 7.2 shows the Producer software task from the producer/consumer example introduced in Section 6.4. The program starts with an infinite while-loop that first performs some calculations that take approximately 50.0 milliseconds. It is followed by a call of the `put(...)` method on the output port of type `osss_port<osss_rmi_if<FIFO_if<Packet> > >`. Before the while-loop is repeated, another calculation that takes approximately 10.0 milliseconds is performed.

```
1  class Producer : public osss_software_task {
2  public:
3    // connection to the "outside world"
4    osss_port<osss_rmi_if< FIFO_if<Packet> > > output;
5
6    // constructor of the software task
7    OSSS_SW_CTOR(Producer) { }
8
9    // the entrance of the software task
10   void main()
11   {
12     Packet p;
13     while(true) {
14       OSSS_EET(sc_time(50.0, SC_MS)) {
15         // some calculations that take approx. 50.0 milliseconds
16         ...
17       }
18
19       // calling the put method on the Shared Object bound to output
20       output->put(p);
21
22       OSSS_EET(sc_time(10.0, SC_MS)) {
23         // some calculations that take approx. 10.0 milliseconds
24         ...
25       }
26     }
27   }
28 };
29
30 OSSS_DEFINE_SW_TASK(Producer);
```

Listing 7.2: OSSS Software Task using an OSSS Port for hardware/software communication

The Estimated Execution Times (EETs) have been utilized to reflect the execution time on the target processor during the simulation on the host system. Therefore, they become meaningless when the code is compiled for the target processor itself. Listing 7.3 shows the empty EET macro that is used to eliminate all EET macros from the software code that has been used during simulation.

Even on the target processor we need to provide a defined entry point for the program. This is usually done by the int `main()` or int `main( int argc, const char* argv[] )` function as defined in the C++ ISO standard. This "real" main and the start of the `main()` method inside the `osss_software_task` is performed by the code behind the `OSSS_DEFINE_SW_TASK( _task_class_ )` macro as shown in Listing 7.3.

```
1  // EETs are not needed when software runs on the target processor
2  #define OSSS_EET(duration)
3
4  namespace osss {
5    namespace osssi {
6      void osss_software_task_start_helper(osss_software_task* task) {
7        task->start_main();
8      }
9    }
10 }
11
12 #define OSSS_DEFINE_SW_TASK( _task_class_ ) \
13   _task_class_ sw_task(#_task_class_); \
14   int main() { \
15     osss :: osssi :: osss_software_task_start_helper(&sw_task); \
16     return EXIT_SUCCESS; \
17   }
```

Listing 7.3: Macros for porting an OSSS Software Task from host simulation to the target processor

All methods that are implemented in hardware (i.e. implemented in Shared Objects) are only allowed to carry parameters of data types defined in the OSSS hardware/software intersection. The following tables define the data types that can be used for hardware/software communication.

Type	Size in bit
bool	sizeof(bool)·BYTE_SIZE
char	BYTE_SIZE
unsigned char	BYTE_SIZE
signed char	BYTE_SIZE
wchar_t	sizeof(wchar_t)·BYTE_SIZE
(signed) short	sizeof(short)·BYTE_SIZE
unsigned short	see above
(signed) int	sizeof(int)·BYTE_SIZE
unsigned int	see above
(signed) long	sizeof(long)·BYTE_SIZE
unsigned long	see above
(signed) long long	sizeof(long long)·BYTE_SIZE
unsigned long long	see above
float	sizeof(float)·BYTE_SIZE
double	sizeof(double)·BYTE_SIZE
long double	sizeof(long double)·BYTE_SIZE

Table 7.2: Supported built-in C++ data types

Table 7.2 defines the supported built-in C++ data types. When transferring data between hardware and software it must be ensured that the interpretation of the type is the same. For the interpretation of data types three properties have to be considered:

1. the size in bit

2. the byte ordering (big- or little-endian)
3. the alignment (for compound data types)

When serializing data in the OSSS RMI protocol we do not explicitly consider the alignment, because all data members of a serializable object are written one after another into a bit vector. Since there is no padding between data members no special interpretations of the alignment are necessary.

The byte ordering is processor dependent. Since the byte ordering in network packets is always big-endian (also called *network order*) we use this order in the `osss_serialisable_object`. The necessary conversions when the software is executed on a little-endian processor are done automatically by the serializable object.

The size of each built-in C++ data type is compiler dependent and cannot be influenced by the designer. Because of the insufficiencies in the ISO C++ standard concerning the definition of the fundamental types (cf. [14] section 3.9.1) it is up to the compiler to define the exact size of each built-in type. For this reason sizes in Table 7.2 are defined with respect to the compiler by using the `sizeof(...)` operator that returns the number of bytes its operand occupies in the memory. In most processor architectures known to the author, the size of a byte (`BYTE_SIZE`) is 8 bit. For some DSPs `BYTE_SIZE` is 16 bit.

Type	Size in bit	Remark
sc_bit	1	
sc_bv<int W>	W	
sc_logic	2	
sc_lv<int W>	2·W	
sc_fix	sizeof(sc_fix)·BYTE_SIZE	Not synthesizable
sc_fixed<int W, ...>	W	
sc_fix_fast	sizeof(sc_fix_fast)·BYTE_SIZE	Not synthesizable
sc_fixed_fast<int W, ...>	$W \in [1 \cdots 53]$	Not synthesizable
sc_ufix	sizeof(sc_ufix)·8	Not synthesizable
sc_ufixed<int W, ...>	W	
sc_ufix_fast	sizeof(sc_ufix_fast)·8	Not synthesizable
sc_ufixed_fast<int W, ...>	$W \in [1 \cdots 53]$	Not synthesizable
sc_bigint<int W>	W	
sc_biguint<int W>	W	
sc_int<int W>	$W \in [1 \cdots 64]$	
sc_uint<int W>	$W \in [1 \cdots 64]$	

Table 7.3: Overview of SystemC data types

Table 7.3 shows the SystemC data types that are supported for hardware/software communication. In general all unconstrained types are not supported. This is also conforming with the SystemC synthesizable subset [33].

Type	Size in bit
osss_enum<class Enum>	$\lceil log_2(\text{max_value(Enum)} + 1)\rceil$
osss_array<class ElementType, unsigned int Size>	sizeof(ElementType)·Size·BYTE_SIZE
osss_serialisable_object	osss_serialisable_object::m_bit_vector.size()

Table 7.4: Overview of OSSS data types

Table 7.4 gives an overview of the OSSS data types for hardware/software communication. The `osss_enum<class Enum>` type is a serializable replacement for the C++ built-in enumeration type. It is just a serializable wrapper class that takes a C++ `enum` as template parameter. The

resulting size in bits is calculated by the formula given in Table 7.4, whereas the max_value(...) function returns the biggest value representation of the enumerators.

The osss_array<class ElementType, unsigned int Size> type is a serializable replacement for the C++ one-dimensional array type. It takes the amount of Size elements of ElementType type. It implements the operator[](...) for accessing the elements of the osss_array<...> in the common way.

As already described in Section 6.5.2 the osss_serialisable_object is the base class for all user-defined data types that need to be used for hardware software communication. The size of an object derived from osss_serialisable_object is defined by the sum of the size of all its serializable members. The resulting size is determined by the size of the bit vector of the serializable object that stores all members added by the serialise() method.

7.6.4.2 RMI Layer

Since a software task can only be the initiator of a communication only the initiator part of the RMI is implemented in the OSSS RMI protocol of the OSSS software library.

The osss_port<osss_rmi_if<IF> > shown in Figure 7.19 represents the interface between the OSSS Software API and the OSSS RMI protocol stack.

The designer has to supply the osss_rmi_if<...> container for each interface implemented by the user-defined class inside a Shared Object.

This interface has to contain stubs for all methods of the type of the interface class. The stubs are generated by the OSSS_METHOD_STUB(...) and the OSSS_METHOD_VOID_STUB(...) macros. The OSSS RMI layer of the OSSS software library provides the same macros in order to reuse the code of all user-defined osss_rmi_if<...> containers.

The communication of the MicroBlaze processor with user-defined hardware blocks is performed through the OPB. Therefore the RMI layer of the OSSS software library provides the osss_rmi_channel<xilinx_opb_channel<...> >::client_if interface only. As a consequence the *Channel Layer* interface needs to provide the osss_abstract_channel<...>::master_if interface only.

In general the implementation of the RMI layer of the OSSS software library is analogue to the implementation of this layer in the OSSS simulation library. This layer of the OSSS software library is implemented with the intention to be as portable as possible between at least all cross-compilers using a gcc compiler front-end.

7.6.4.3 Channel Layer

The *Channel Layer* of the OSSS software library shown in Figure 7.19 is the most processor specific layer of the RMI protocol stack. The *Channel Interface Layer* is used to encapsulate the processor dependent implementation from the portable (or processor independent) RMI layer.

The master_if implemented in the *Channel Layer* performs the translation from the abstract interface using read_blocking(...) and write_blocking(...) methods to the Xilinx MicroBlaze specific input/output mechanisms.

The chosen output mechanism of the MicroBlaze depends on whether burst transfers are used or not used. As discussed in Section 7.6.2 the OPB that is used to connect the MicroBlaze to the user-defined hardware can be attached to the processor in two different ways. The first one is to use the native bus interface (IOPB and DOPB) of the MicroBalze. The second one uses the Fast Simplex Link (FSL) to communicate with a dedicated FSL-to-OPB Bridge, which is capable of initiating burst transfers on the OPB.

7.6.4.4 The native OPB Interface

The major drawback of the native bus interface is that it does not support the initiation of burst transfers on the OPB. This limitation results from the technique of memory mapped I/O that is used for the communication with the peripherals/slaves connected to the OPB. Each data transfer to a peripheral is described by a memory access to the specific address it is mapped to. The main advantage of this technique is its simplicity in terms of usage and implementation since no additional communication controller is needed.

```
typedef unsigned char   Xuint8;  // unsigned 8-bit
typedef char            Xint8;   // signed 8-bit
typedef unsigned short  Xuint16; // unsigned 16-bit
typedef short           Xint16;  // signed 16-bit
typedef unsigned long   Xuint32; // unsigned 32-bit
typedef long            Xint32;  // signed 32-bit

#define XIo_In8(InputPtr)  (*(volatile Xuint8  *)(InputPtr))
#define XIo_In16(InputPtr) (*(volatile Xuint16 *)(InputPtr))
#define XIo_In32(InputPtr) (*(volatile Xuint32 *)(InputPtr))

#define XIo_Out8(OutputPtr, Value)  \
    { (*(volatile Xuint8  *)(OutputPtr) = Value);  }

#define XIo_Out16(OutputPtr, Value) \
    { (*(volatile Xuint16 *)(OutputPtr) = Value);  }

#define XIo_Out32(OutputPtr, Value) \
    { (*(volatile Xuint32 *)(OutputPtr) = Value);  }
```

Listing 7.4: Xilinx MicroBlaze specific basic input/output macros

Listing 7.4 shows the Xilinx MicroBlaze specific basic input/output macros. These can be used to access slave components, which are connected to the OPB and are properly mapped in the processor's memory space. The input and output macros are defined in three different granularities for byte (8-bit), half-word (16-bit) and word (32-bit) accesses.

```
bool write_blocking(address_type slave_base_addr, const data_chunk& data) {
  switch(data.size()) {
    case 0:
      return false;
    case 1: {
      Xuint8 byte = data[0];
      XIo_Out8(slave_base_addr, byte);
      break;
    }
    case 2: {
      sc_uint<16> d;
      d.range(7, 0)  = data[0];
      d.range(15, 8) = data[1];
      Xuint16 word = d;
      XIo_Out16(slave_base_addr, word);
      break;
    }
    case 4: {
      sc_uint<32> d;
      d.range(7, 0)   = data[0];
      d.range(15, 8)  = data[1];
      d.range(23, 16) = data[2];
      d.range(31, 24) = data[3];
      Xuint32 dword = d;
      XIo_Out32(slave_base_addr, dword);
      break;
    }
    default: {
      vector<sc_uint<32> > data32 = v32_from_v8(data);
      Xuint32 size = data.size()
      XIo_Out32(addr, size);
      for(unsigned int i = 0; i < data32.size(); i++) {
        Xuint32 dword = data32[i];
        XIo_Out32(addr, dword);
      }
      break;
    }
  }
  return true;
}
```

Listing 7.5: Implementation of the `write_blocking(...)` method using the native OPB interface

Listing 7.5 shows the implementation of the `write_blocking(...)` method. It is defined in the `osss_abstract_chanel<...>::master_if` interface. The minimal addressable size of the Xilinx OPB is a byte and therefore the `data_chunk` is of type `std::vector<sc_uint<8> >`. When the `write_blocking(...)` method is called it first checks the size of the `data_chunk`. If the size of the `data_chunk` is zero no transmission is initiated and `false` (which indicates an error) is returned to the caller of this method. If the size of the `data_chunk` is one, two or four, the corresponding `XIo_Out` macro is called for the given address. If the size of the `data_chunk` is three or greater than four first of all the size is written to the addressed slave. After the size has been submitted the `data_chunks` are written to the same address with a granularity of 32-bit one after another. Using the FSL interface, this kind of transaction is implemented as a burst on the OPB.

7.6.4.5 The FSL Interface

The Fast Simplex Link (FSL) interface of the Xilinx MicroBlaze processor has been introduced in Section 7.4. Since the FSL is directly mapped into the register file of the MicroBlaze, special put and get instructions are provided. Listing 7.6 shows the blocking variants of the Xilinx MicroBlaze specific FSL access macros[4]. They are provided to encapsulate the assembler code for accessing the FSL specific registers.

```
1  // Blocking Data Read and Write to FSL channel with id
2  #define microblaze_bread_datafsl(val, id)  \
3          asm volatile ("get %0, rfsl" #id : "=d" (val))
4
5  #define microblaze_bwrite_datafsl(val, id)  \
6          asm volatile ("put %0, rfsl" #id :: "d" (val))
7
8  // Blocking Control Read and Write to FSL channel with id
9  #define microblaze_bread_cntlfsl(val, id)  \
10         asm volatile ("cget %0, rfsl" #id : "=d" (val))
11
12 #define microblaze_bwrite_cntlfsl(val, id)  \
13         asm volatile ("cput %0, rfsl" #id :: "d" (val))
```

Listing 7.6: Xilinx MicroBlaze specific FSL access macros

There are two different blocking read and blocking write macros. One macro is used to transfer data (`_datafsl`) and another one to transfer control information (`_cntlfsl`). The `val` parameter represents the transferred data value and the `id` parameter specifies which of the 8 different FSL interfaces is used.

```
1  bool write_FSL2OPB_single(const Xuint32 addr, const Xuint32 data,
2                            const FSL2OPB_transfer_mode transfer_mode) {
3    if ((transfer_mode == FSL2OPB_8)  ||
4        (transfer_mode == FSL2OPB_16) ||
5        (transfer_mode == FSL2OPB_32)) {
6      microblaze_bwrite_cntlfsl(transfer_mode, FSL2OPB_OUT_ID);
7      microblaze_bwrite_cntlfsl(addr, FSL2OPB_OUT_ID);
8      microblaze_bwrite_datafsl(data, FSL2OPB_OUT_ID);
9      return true;
10   }
11   return false;
12 }
13
14 bool write_FSL2OPB_burst(const Xuint32 addr, const data_chunk& data) {
15   Xuint32 size = data.size();
16   vector<sc_uint<32> > data32 = v32_from_v8(data);
17   microblaze_bwrite_cntlfsl(FSL2OPB_BURST, FSL2OPB_OUT_ID);
18   microblaze_bwrite_cntlfsl(addr, FSL2OPB_OUT_ID);
19   microblaze_bwrite_cntlfsl(size, FSL2OPB_OUT_ID);
20   for(unsigned int i = 0; i < data32.size(); i++) {
21     Xuint32 fsl_data = data32[i];
22     microblaze_bwrite_datafsl(fsl_data, FSL2OPB_OUT_ID);
23   }
```

[4]There are also non-blocking versions of these macros that are not further mentioned here.

```
24    return true;
25  }
26
27  bool write_blocking(address_type slave_base_addr, const data_chunk& data) {
28    switch(data.size()) {
29      case 0:
30        return false;
31      case 1: {
32        return write_FSL2OPB_single(slave_base_addr, data[0], FSL2OPB_8);
33      }
34      case 2: {
35        sc_uint<16> d;
36        d.range(7, 0)  = data[0];
37        d.range(15, 8) = data[1];
38        return write_FSL2OPB_single(slave_base_addr, d, FSL2OPB_16);
39      }
40      case 4: {
41        sc_uint<32> d;
42        d.range(7, 0)   = data[0];
43        d.range(15, 8)  = data[1];
44        d.range(23, 16) = data[2];
45        d.range(31, 24) = data[3];
46        return write_FSL2OPB_single(slave_base_addr, d, FSL2OPB_32);
47      }
48      default: {
49        return write_FSL2OPB_burst(slave_base_addr, data);
50      }
51    }
52  }
```

Listing 7.7: Implementation of the `write_blocking(...)` method using the FSL interface

Listing 7.7 shows the implementation of the `write_blocking(...)` method using the FSL interface that is connected to the FSL2OPB Bridge. The `write_FSL2OPB_single(...)` method is used to initiate single cycle transfers on the OPB. The `FSL2OPB_transfer_mode` is used to specify the granularity of the transfer, which can either be 8-bit, 16-bit or 32-bit. The communication with the FSL2OPB Bridge works as follows. The transfer starts by the submission of the transfer mode followed by the slave base address by using the `microblaze_bwrite_cntlfsl(...)` macro. After this control information the data is written by using the `microblaze_bwrite_datafsl(...)` macro.

The `write_FSL2OPB_burst(...)` method is used to initiate a burst transfer on the OPB. It starts by setting the transfer mode to `FSL2OPB_BURST` and by submitting the base address of the slave by using the `microblaze_bwrite_cntlfsl(...)` macro. Before starting with the data transfer the length of the burst in number of bytes is transmitted. Afterwards the data is written by using the `microblaze_bwrite_datafsl(...)` macro in a for-loop.

7.6.5 Software Cross-Compilation

The synthesis flow for the hardware and the software part of an OSSS design are separated from each other. The user-defined hardware of the design is processed by *Fossy* (see Section 7.7.1). The user-defined software of the design needs to be cross-compiled for the processors defined by the hardware platform. Up to now, only the MicroBlaze soft processor core is supported.

For the cross-compilation the MicroBlaze gcc (`mb-gcc`) using a modified `gcc` 3.4.1 is used. The Xilinx EDK provides the complete GNU tool chain for the MicroBlaze processor. The input for the MicroBlaze cross-compiler is the entire code that is used inside an `osss_software_task`. When compiling an `osss_software_task` for the MicroBlaze the OSSS *Software Library* needs to be included. It contains the OSSS software API and the OSSS RMI protocol stack describe before.

As shown in Figure 7.27 the OSSS *Software Library Generator* is used to add design specific code to the OSSS *Software Library*. This design specific code consists out of unique IDs (client ID, object ID, method ID). This information is necessary to set up a working hardware/software communication. Both the OSSS RMI protocol stack running on the MicroBlaze and the implementation of the hardware interface performed by *Fossy* need to share this specific information.

After the `osss_software_task` (including the OSSS Software Library) has been compiled by the `mb-gcc`, it is linked with the Xilinx library. The Xilinx library is generated and compiled by the Xilinx EDK Library Generator by using the `mb-gcc` as well. It includes code for performing I/O with the MicroBlaze processor and communication with Xilinx specific peripherals (e.g. a `printf(...)` function and `std::cout` is provided for sending strings to the UART).

The linking is controlled by a Linker Script that is needed because the MicroBlaze runs in “stand-alone” mode, i.e. without an operating system. The Linker Script defines the memory layout of the application. It defines where the different sections for services like memory allocation and object destruction are mapped in the memory. Additionally it defines the stack and the heap size. Linker Scripts for program execution from internal (Block RAM only) and/or external (on board DDRAM) memories are provided.

After linking has been performed an executable file in the Executable and Link Format (ELF) is generated. When running an application from the internal Block RAM the Xilinx Bitstream Initialiser is used to initialize the Block RAMs with the context of the ELF file. After initializing the `system.BIT` with ELF data from the software compilation the result is a bitstream with initialized Block RAM resources, called `download.BIT`. This configuration file can be downloaded to the FPGA via JTAG by using the Xilinx iMPACT tool. When the resulting application is too big to be executed from the block RAM only, it needs to be loaded into the external RAM. This can be performed by a special boot loader via the RS232 port or by the Xilinx Microprocessor Debugger (`XMD`, with `XDM_stub`) in conjunction with the GNU Debugger (GDB) via JTAG.

7.7 Custom Hardware Synthesis

The synthesis of custom hardware is performed by *Fossy* (**F**unctional **O**ldenburg **S**ystem **SY**nthesiser). The hardware interface synthesis affects two entities of a design, namely Shared Objects and hardware clients of Shared Objects. The mapping of the *Application Layer* to the *Virtual Architecture Layer* guides the hardware interface synthesis process by means of well defined communication refinement rules.

7.7.1 *Fossy*

Figure 7.20 gives an overview of the *Fossy* tool chain. *Fossy* is written in the pure functional programming language Haskell [215].

Since both, the internal *Fossy* data structure and the intermediate format of the front end are based on the same ISO/IEC C++ standard [14] the implementation of the adaptation layer is straightforward. The EDG front end parser (called cc2cil) [217] is used and augmented with a thin layer converting the front end specific intermediate format to XML (called cil2xml). Large parts of the logic of this conversion layer are automatically generated from the Haskell type structure.

Haskell values conforming to the type system representing the C++/SystemC/OSSS grammar are written to and read from XML files. The DTD describing the structure of the XML is derived from the type system automatically. The human readable XML data is the input for *Fossy* and is open to a range of existing XML tools.

Besides the OSSS design on *Application Layer* information from the *Virtual Target Architecture Layer* are needed by *Fossy*. As described in Section 7.5 the architectural context extraction step is used to generate architectural context information (ACI) that are stored and forwarded to *Fossy* in an XML format.

The ACI file contains the following information:

1. The channel bindings, i.e. the mapping of the communication links to OSSS-Channels. This implies the number of clients to a Shared Object.

2. Transactor specific parameters e.g. channel bit width of a point-to-point channel and each client’s address range at the Object Socket. Note that the address ranges imply the Shared

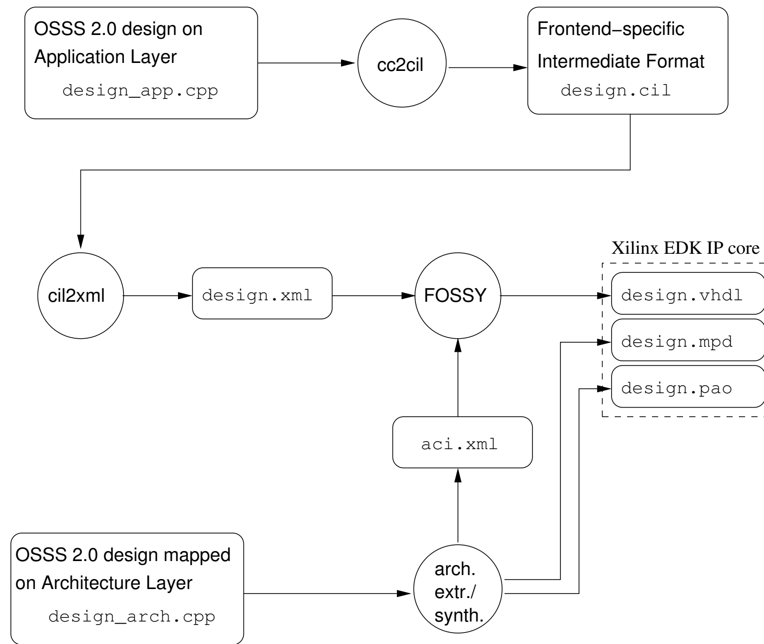


Figure 7.20: *Fossy* high-level synthesiser tool chain [44]

Object ID and the client IDs, since each client has a unique address range for each of its serving Shared Objects.

3. Method IDs for each method of a Shared Object.
4. The layout of the transmitted, i.e. serialized data.

The output of *Fossy* is a synthesizable VHDL description of the user-defined hardware part of the OSSS design. For using this output in the Xilinx back-end synthesis flow it needs to be "imported" to the Xilinx EDK project. As shown in Figure 7.2 the *Fossy* synthesis output is stored in the user repository of the EDK project and thus constitutes a part of the overall OSSS design architecture. From the Xilinx EDK's point of view the output generated by *Fossy* is treated like a 3rd party IP component.

7.7.2 Synthesis Phases

Fossy is a high-level synthesizer that transforms an OSSS/SystemC description into synthesizable VHDL code. Compared to other high-level synthesis tools, *Fossy* does *not* perform automatic scheduling of an unscheduled purely sequential C/C++ algorithm. In OSSS the scheduling of the entire design is performed either statically by the designer using appropriate wait statements (also called behavioral RTL) or dynamically by using the Shared Object scheduling capabilities.

In the following, we are going roughly through *Fossy*'s synthesis phases and their associated design transformations (ref. Figure 7.2).

7.7.2.1 Elaborator

The purpose of the *Elaborator* module is to infer the resulting SystemC design from a raw C++ translation unit as represented by the front end. More specifically this means to:

- identify and distinguish modules, ports, black-box IP components and signals (including signals from user-defined classes)
- identify and distinguish processes (`SC_METHOD` and `SC_CTHREAD`).
- find the top-level module (i.e. the module derived from `osss_system`)
- detect SystemC data types

7.7.2.2 Channel Synthesis

The Channel Synthesis step in *Fossy* is the replacement of the Application Model communication links with channels enabling a signal-based and synthesizable communication. The information about the communication link to OSSS Channel binding has been generated during the Communication Structure Analysis step of the Architecture Extraction process. The channel binding information is propagated to *Fossy* by an ACI file.

Figure 7.21 demonstrates how the communication link to channel binding information is used for the transformation of the communication structure and the channel instantiation. On the left side, a hierarchical OSSS Application Layer Design is shown. Client processes (can either be Software Tasks or Hardware Modules), denoted by `C1` to `C4` are connected to Shared Objects (`SO1` & `SO2`). Dashed connections are abstract communication links (port to interface bindings) while solid connections are signal-level port-to-port bindings. The hierarchical module containing `C2` to `C4` has an additional process (denoted by the circular arrow) that is a client of `SO1`. All connections that cross the hierarchy boundary are using ports, denoted by small square symbols.

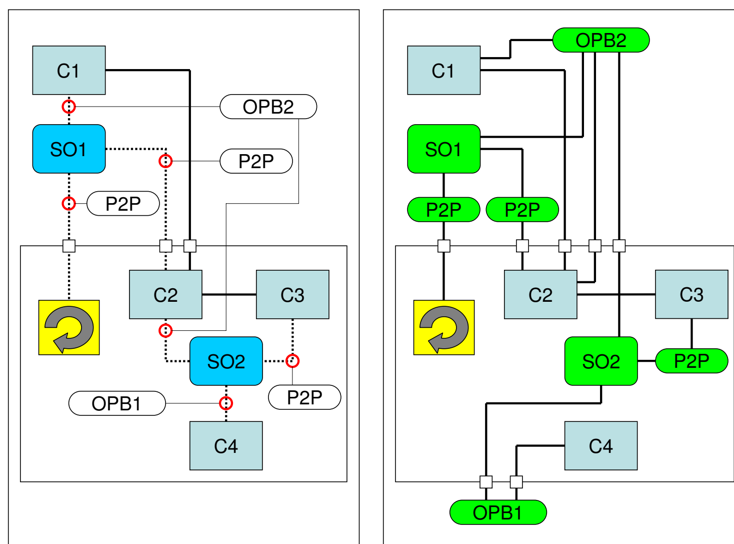


Figure 7.21: Using of the communication mapping information for Channel Synthesis

All communication links on the left side are annotated with their channel bindings from the ACI. In this example, the communication links are mapped onto two OPB and three point-to-point (P2P) channels.

On the right side of Figure 7.21 the design after channel synthesis is shown. All communication links have been implemented through channels and all connections to channels are low-level signal communications. As one can see, two rules are applied during channel instantiation:

- The channel is instantiated on the lowest hierarchy level possible with respect to its communication partners position within the design hierarchy.
- Channels whose internal structure is not generated by *Fossy* (IP channels) like the Xilinx OPB, need to be moved to the top-level hierarchy.

7.7.2.3 Shared Object Synthesis

The purpose of this synthesis step is the conversion of Shared Objects into “ordinary” hardware modules. Figure 7.22 demonstrates the structural representation of a Shared Object after synthesis. The shown Shared Object has two different physical interface (i.e. Object Socket connections to different OSSS Channels).

Both *Interface Blocks* `IF1` and `IF2` consist of a channel protocol specific part and an RMI protocol specific part. The connection of both interface blocks to the channels is just a set

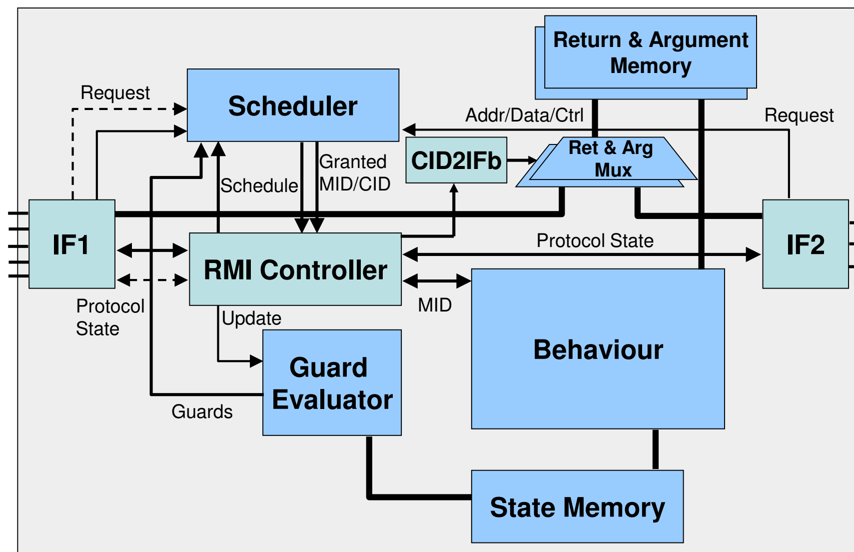


Figure 7.22: Example of a Shared Object's representation after synthesis

of wires. The channel protocol specific part drives these wires according to the channel's communication protocol. The communication with the internal structure of the Shared Object is performed through the RMI protocol.

The *Scheduler Block* implements the chosen scheduling algorithm. The *Guard Evaluator* checks whether the guard condition associated with the requested guarded method call evaluates to true. Therefore it needs access to the *State Memory* of the Shared Object's *Behavior*.

The *Argument & Return Parameter Memory* is used to store incoming and outgoing parameters of guarded method calls. These memories as well as the state memory either are implemented as registers or dedicated Block-RAMs.

The *Behavior Block* contains the method bodies of all guarded methods form the user-class inside the Shared Object.

Finally, the *RMI Controller* handles the RMI protocol with the connected clients and drives the Shared Object's internal protocol.

More details of the Shared Object synthesis are described in Section 7.8.

7.7.2.4 Class Synthesis

During class synthesis the following steps are performed on the translation unit:

- Removal of unused classes. Definitions of classes which are never instantiated are removed from the AST.
- Inlining of all global variables, which must be either `const` or built-in-initialized. Globally shared variables are not allowed in a hardware design, because of its unrestricted access from different processed and possibly non-deterministic behavior due to race-conditions. However global constants and built-in-initialized global variables are becoming members of all classes using them.
- `Private` and `protected` access protectors are substituted by `public`. Proper access to class members and member functions is checked during compile-time. Thus in the internal representation we can omit access protection and set all to `public`.
- Transformation of constructors to functions.
- Class flattening: Conversion of classes to baseless classes. All base classes are mapped to member attributes and all base class accesses are changed to access the base member attribute.

- Loop normalization: Conversion of for-loops and do-loops to while-loops.
- Transformation of methods to functions: Every method is transformed to a function that receives a pointer to the object instance as additional first parameter. All method calls are transformed to function calls. Methods defined outside the *Fossy* namespace (e.g. SystemC methods like `read()` and `write()`) are not transformed in this step.
- Removal of pointer variables which are created during constructor and method elimination. In addition, pointers are converted to references and the "address of" (&) and "dereference" (*) operators are eliminated[5].
- Finite State Machine (FSM) transformation: The purpose of this step is to transform implicit state machines (`SC_CTHREAD`s with embedded `wait(...)`s) into explicit ones, i.e. `SC_METHOD`s with explicit state variables. The basic idea is as follows: Each wait is associated with a state: When a clock edge occurs, an `SC_CTHREAD` awakes from a certain wait and runs into another wait. In order to create an `SC_METHOD` which behaves like the `SC_CTHREAD` it must store the wait state where to start next in its state variable. In this state it performs its work and sets the next state. The FSM transformation looks at the waits of the `SC_CTHREAD` and finds the trace of statements which may be executed until the next wait is reached. Figure 7.23 shows a body of an `SC_CTHREAD` and all traces from one wait to another. Each trace results in an explicit state after transformation.
- Union synthesis: Elements of unions are analyzed for the "largest" element. A struct with a bitvector of the size of the largest element is used to represent the union. Access function for all union elements are added and appropriate bitvector casts are added to write and read the data from the single bitvector representing the union.
- Synthesis of bitvector casts.
- Localization of variables in functions: Moves all local variable declarations to the beginning of the function (C-like). Constants are converted to variables in order to separate the initialization from the declaration. However, constant expressions should also move to the front to keep the `const` qualifier and get constants in output later on.
- Fixup local variables in processes: Moves all local variables of `SC_METHOD`s into the local variable section of the class.
- Class to struct transformation: All classes are replaced by structures and functions on the data members of this structure.
- Removal of unused functions and unused enumerations.

7.7.2.5 Integer Type Synthesis

The integer type synthesis converts arithmetic expressions from C++ and SystemC into a form, which is also valid in VHDL. The goal of the synthesis is to generate functional equivalent C++ and SystemC integer type arithmetic in VHDL.

The supported integer types in SystemC are

1. `sc_int<W>` signed and `sc_uint<W>` unsigned integers of bitwidth `W`
   - Must use native C++ data types internally
   - Must support at least 64 bits
   - Maximum bitwidth is implementation-dependent
   - Allow high simulation speed due to the native internal representation
2. `sc_bigint<W>` signed and `sc_biguint<W>` unsigned integers of bitwidth `W`
   - Must support any bitwidth

[5]In OSSS pointers are only allowed for sub-module instantiation. No user-level pointers are supported. For more details see Appendix F

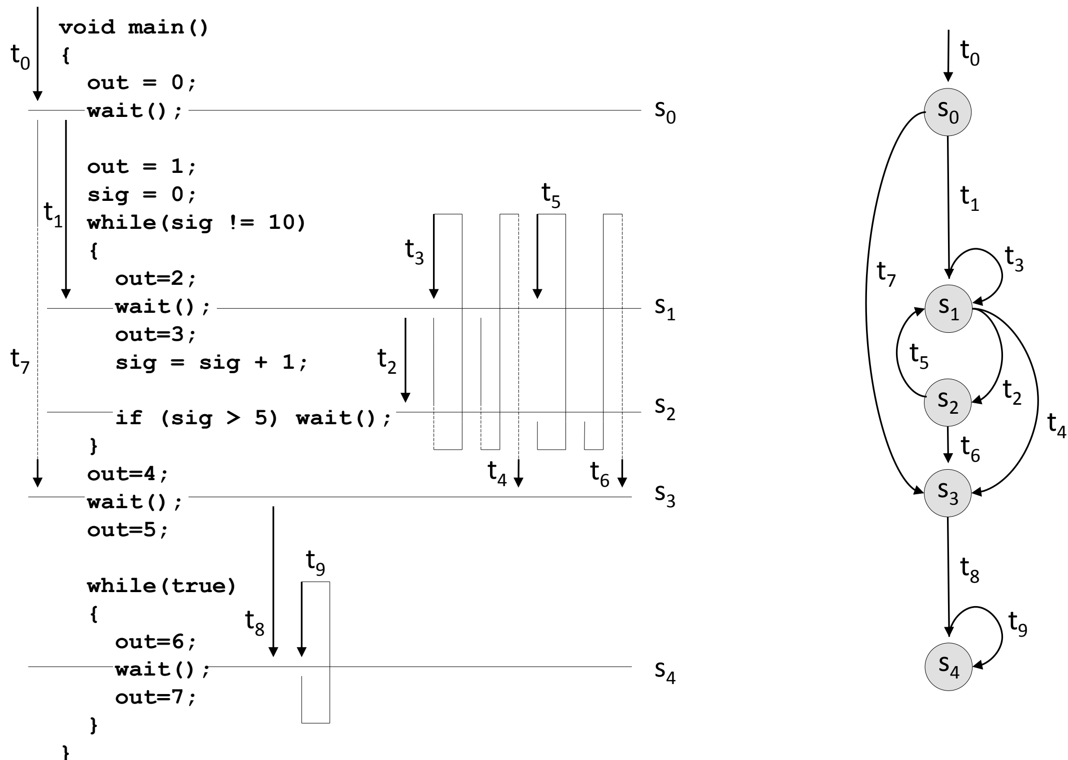


Figure 7.23: Implicit to explicit state machine transformation

- Flexibility has higher overhead and decreases simulation speed

In a first step all C++ internal integer types are mapped to `sc_int<W>` and `sc_uint<W>`. For used bitsizes see Table 7.2. And in a second step `sc_(u)int<W>` and `sc_big(u)int<W>` are transformed into an equivalent VHDL representation including integer arithmetic.

In SystemC arithmetic is performed using a hidden base classes without visible bitwidths. This invisible automatic sign extensions may cause overflow, e.g.

```
sc_uint<64> x = 18446744073709551615ul; // max uint_64
        x + x = 18446744073709551614    // overflow
sc_bigint<70>(x) + sc_bigint<70>(x) = 36893488147419103230  // correct result
```

The result's bitwidth of a subtraction of two unsigned integers in SystemC is bigger than expected, because subtraction always promotes its operands to signed. Division in SystemC needs an extra bit to catch MAX_NEG/-1. In addition, the `operator«` (shift left) is unbound on signed integers in SystemC, e.g.

```
(sc_bigint<4>(1) << 5).length() = 9
(sc_bigint<4>(1) << 50).length() = 54
```

Table 7.5 gives a comparison of SystemC and VHDL integer arithmetic. To overcome these differences *Fossy* uses a simplified internal representation consisting of the following three basic data types: SIGNED, UNSIGNED, BITVECTOR with the following mapping:

`sc_int<W>`, `sc_bigint<W>` → SIGNED
`sc_uint<W>`, `sc_biguint<W>` → UNSIGNED
`sc_bv<W>` → BITVECTOR

All values have bit indices from `MSB:(width-1) downto LSB:0` and all range expressions have type BITVECTOR. Size changes (implicit/explicit casts, arithmetic extensions) become

SystemC	VHDL
Division needs special care because of extra bit to catch MAX_NEG/-1	Division needs special care because of extra bit to catch MAX_NEG/-1
Automatic argument expansion in arithmetic	No automatic argument expansion in arithmetic
Result bitwidths big enough for result values	Result bitwidths of add/sub unchanged
Unbounded shift left	Sane shift left
Ranges implemented with lots of helper classes	Ranges may have signs, ranges are not expressions

Table 7.5: Comparison of SystemC and VHDL integer arithmetic

explicit RESIZE expressions. Results of arithmetic operations have types/widths following SystemC semantics:

- promote to signed when there is a signed type involved anywhere,
- use a types wide enough to hold the results and
- shift-left needs an explicit result bitwidth.

The mapping of this intermediate representation to VHDL data types[6] is as follows:

SIGNED → SIGNED
UNSIGNED → UNSIGNED
BITVECTOR → STD_LOGIC_VECTOR

With this mapping:

- Results of range expressions need adjustment to `(width-1) downto 0`.
- Results of range expressions need sign adjustment.
- Casts/arithmetic extensions become calls to RESIZE function.
- SystemC semantics implemented in library using standard VHDL
- Use of functions results in prefix notation

The following SystemC integer calculation:

```
sc_int<64> result;
sc_uint<5> x1 = "0b11111";  // 31
sc_uint<6> x2 = "0b111111"; // 63
result = (x1 + x2) * 4;     // 376
```

gets transformed into:

```
variable result : SIGNED(63 downto 0);
variable x1 : UNSIGNED(4 downto 0) := "11111";
variable x2 : UNSIGNED(5 downto 0) := "111111";

result := FOSSY_RESIZE(
            FOSSY_MUL(
              FOSSY_ADD(x1, x2),
                TO_SIGNED(4, 32)),
          64);
-- result = 376
```

where the naïve mapping to the default VHDL functions would generate a wrong result:

```
result := RESIZE( SIGNED("0" & (x1 + x2)) * TO_SIGNED(4, 32), 64);
--  result = 120
```

[6] using `IEEE.STD_LOGIC_1164.all` and `IEEE.NUMERIC_STD.all`

7.7.2.6 Delaborator

The delaborator's task is to transform an (elaborated) OSSS design into a form on which code generators work directly, i.e. without any further complex transformations. Furthermore, it performs name shaping to ease readability of the output code and traceability to the OSSS Application Layer input design.

7.7.2.7 Code Generator

Fossy has two different code generators that write the nodes of the AST either to synthesizable VHDL or to equivalent RTL SystemC code. The information about the design partitioning gained during the Architecture Exploration step is used here to generate appropriate VHDL files for each user-defined hardware partition.

For using this VHDL output in the Xilinx back-end synthesis flow it needs to be copied/integrated into the Xilinx EDK project structure that has been generated at the end of Architecture Extraction phase.

7.8 Shared Object Hardware Synthesis

In this section, we describe the synthesis process of the hardware part which is necessary for a hardware/software or hardware/hardware communication.

In OSSS method-based communication between software and hardware or hardware and hardware is uniformly done via the RMI protocol and the callee of such a communication must always be a Shared Object (see Figure 7.24). Therefore, the hardware interface synthesis regarding the Shared Object is independent of whether the caller is a Hardware Module or a Software Task. Hence hardware interface synthesis affects two entities of a design, namely Shared Objects and hardware clients (i.e. Hardware Modules). These two cases will be described in the following subsections.

7.8.1 Overview

Figure 7.24 shows the domain of the hardware interface synthesis for hardware/software communication. A Shared Object which is wrapped by an `osss_object_socket` is connected to a `xilinx_opb_channel` on the left and an `osss_simple_point_to_point_channel` on the right side. Therefore, the Shared Object (or more precisely the `UserClass` inside the Shared Object) can be called by any master connected to the OPB (this includes software running on a MicroBlaze) and by the hardware client connected to the point-to-point channel. The `osss_object_socket_port` along with the transactor of the `osss_rmi_channel` and the transactor of the `xilinx_opb_channel` or the `osss_simple_point_to_point_channel` describes the interface of the Shared Object. For a more detailed description of the elements shown in Figure 7.24 refer to Section 6.5.2.

Previous research work on the hardware synthesis of Shared Objects [135, 142, 97] focused on the optimization of Shared Objects for HW/HW communication using a fixed protocol and communication channel (full parallel access). In this thesis, the new requirements (derived from Section 2.4) necessitate an extension of the existing synthesis strategy in the following way:

- The Shared Object shall be separated from the communication channel's media access and physical layer of the connected clients.
- A Shared Object shall be connected to existing buses (for the communication with the CPU/SW).
- To enable Hardware/Software communication a hardware independent communication protocol, called Remote Method Invocation (RMI) shall be handled by a Shared Object.
- Efficient storage of the Shared Object's state in a hardware/software independent representation.

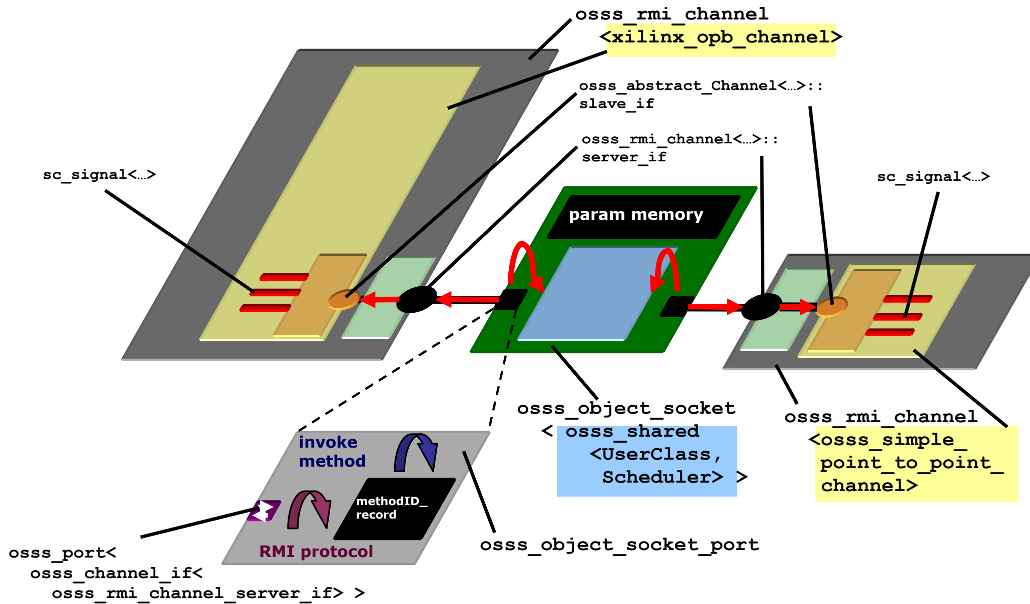


Figure 7.24: The considered hardware interface part for HW/SW and HW/HW communication

The extended, i.e. hardware/software-enabled, synthesized Shared Object is shown in Figure 7.25 (in dark blue displayed the new parts of the Shared Object implementation). The figure follows the producer/consumer example (see Section 6.4) with a software client running on a CPU connected via a bus to the interface *Bus Server IF* (on the left side) and two hardware clients directly connected via a point-to-point connection to *P2P Server IFs* (on the right side). In other words, the interfaces are the connections of the Shared Object to its clients. These interfaces are mainly necessary due to the first requirement (connection to existing buses). The controller within the Shared Object handles the RMI protocol and manages concurrent requests with the help of the scheduler.

The efficient storage requirement, which is a direct consequence of the requirements from Section 2.5 is addressed by the usage of RAM resources: The arguments passed to the guarded methods and the corresponding return values are stored in the argument RAM. The Shared Object's state is hold in the state RAM. For the selected target platform, RAMs are mapped to the Block-RAM (BRAM) resources available on the FPGA. As can be seen in Figure 7.25 the Block-RAMs are used in a dual port configuration. In the case of the argument RAM it is necessary to add an additional multiplexer if there is more than one interface which needs access to the argument RAM.

The synthesis of the scheduler, the guard evaluator and the behavior is based on the work in [135, 142, 97]. The most significant change is new organization of the interface blocks, the integration of the RMI Controller, the access of the *guard evaluator* and the *behavior* to the *state, return & argument memory* access.

The generated hardware structure only depends on the total number of clients, but neither on their types, i.e. hardware or software nor on their specific connection to the Shared Object. The *Communication Channel Interface* ring around the Shared Object's inner structure abstracts from the concrete communication channel's media access and physical layer. For Xilinx Bus IP (e.g. OPB and PLB) a configurable Intellectual Property Interface (IPIF) is used to keep the *Bus Server IF* separated from bus protocol specific details. The Bus Server Interface only requires a simple register and address access to the bus. All other details are hidden by the IPIF block. For more details regarding the IPIF, see Section E.3.7.

The inner structure of a certain interface block (i.e. the interface blocks which either connect

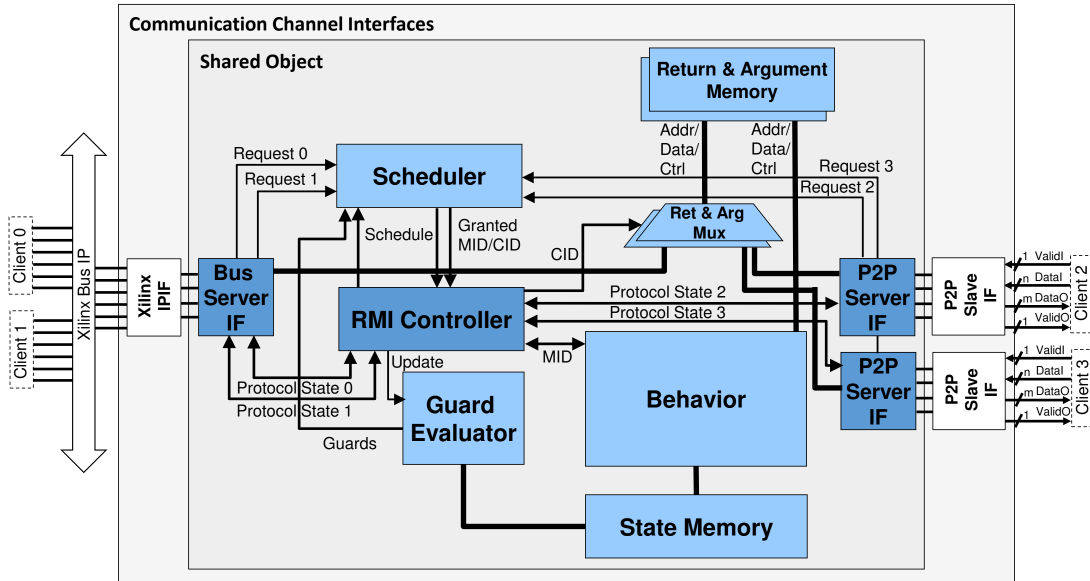


Figure 7.25: Structure of a synthesized Shared Object

the Shared Object to a shared bus or dedicated point-to-point channel) only depends on a few parameters like the number of clients and their corresponding addresses (for bus interface only). For a point-to-point channels the only parameter is the bit width. Therefore, a prototypical synthesis approach is to instantiate pre-designed and properly parametrized hardware structures at the interfaces.

For the synthesis of the hardware client interface a predefined macro that represents a combination of the RMI- and the master transactor is used.

For the first synthesis tool prototype, only certain predefined OSSS Channels over which RMI can be performed, are allowed. Hence, the synthesis tool needs to know on what kind of predefined channel a communication link is mapped, but it does not process the channel-internal implementation.

7.8.2 RMI Controller

The controller is the part of the Shared Object, which manages the RMI protocol and triggers the scheduler, the guard evaluator and the behavior process. For each client, the controller stores the current protocol phase, which is one of the following four:

- `IssueRequest`: client is allowed to request the execution of a guarded method
- `SendParameters`: client must supply the arguments to the method it requested
- `WaitForCompletion`: client must wait for the execution of the method to complete
- `ReadReturnValue`: client must read back the result of the method

When the Shared Object is reset, the controller triggers the constructor method in order to initialize the object's state. Afterwards, it triggers the guard evaluator to update the guards according to the current state. When the guards are up to date it signals the scheduler to select one of the method requests. After the scheduler selected a certain request, it signals the method ID and the corresponding client ID to the controller. The controller then advances the protocol phase of the selected client from `IssueRequest` to `SendParameters` and switches the Ret- & Arg-RAM multiplexer according to the client ID. When it is informed by the interface block that all parameters have been submitted, it starts the behavior process by supplying the ID of the requested method. Simultaneously, to starting the method it sets the calling client's protocol phase to `WaitForCompletion`. When the behavior process signals the completion of the method, the controller advances the state to `ReadReturnValue` and waits until the interface

block signals that the phase is completed. Finally, it sets the protocol phase `IssueRequest`, triggers the guard evaluator and the scheduler and a new cycle begins.

The naive synthesis of the controller is straight forward: basically it has to provide a state machine which creates Shared Object-internal the control signals as described in the previous scenario. The generated hardware structure only depends on the total number of clients, but neither on their types, i.e. hardware or software nor on their specific connection to the Shared Object.

7.8.3 Interface Blocks

The interface blocks *Bus & P2P Server IF*, as shown in Figure 7.25 are the connections of the Shared Object to a certain signal-level bus or point to point connection. Clients can access the Shared Objects only via these connections. There may be multiple clients behind one interface block. For instance, if a Bus Server IF is connected to a bus like the OPB, requests from different clients may be sent to the Shared Object. In this case, there are multiple connections between the interface block and the scheduler as well as between interface block and controller, namely one per client attached to the interface. This case is indicated by the two request signals `Request 0` and `Request 1` for client 0 and 1 in Figure 7.25.

The signal-level interface of a Shared Object's interface block to the outside world can be derived from the (slave) transactor of the connected OSSS Channel (see Section 6.5.3). This information is only available on the Virtual Target Architecture layer where communication links are actually mapped to OSSS Channels. The signals of the interface block into the Shared Object are independent of the bound channel and only depend on the number of clients handled by the particular interface block. The signals into the Shared Object are:

- a request line to the scheduler which indicates the requested method via its unique method ID (MID)
- data, address and control lines for the argument RAM connected to the multiplexer
- an input from the controller which indicates the current protocol phase and
- an output to the controller in order to indicate it can proceed with the next phase

Furthermore, an interface block needs a state machine for proper cooperation with the controller and some control logic in order to create correct control signals for the argument RAM. A description of the cooperation between the interface block and the controller from the interface block's point of view can be found at the end of this section. A description of the same scenario from the controller's point of view can be found in the next section.

The inner structure of a certain interface block, e.g. for an interface block which connects to an OPB, is relatively fixed. It only depends on a few parameters like the number of clients and their corresponding addresses. The same is true for point-to-point channels where the only parameter is the bitwidth. Therefore, a prototypical synthesis approach is to instantiate pre-designed and properly parametrized hardware structures.

Example:
Suppose a SW task wants to call a guarded method of a Shared Object. It issues its request by writing the requested method ID to a certain memory location (which is created by the software interface synthesis). In the `IssueRequest` state the interface block detects this memory access and forwards the requested method ID to the scheduler. The controller will proceed with the `SendParameters` state (in the case the method has parameters). When the interface block detects the `SendParameter` state it will monitor the bus and listen for writes to the argument RAM. When the interface block detects that all parameters have been transferred to the argument RAM it signals the controller to proceed with the next phase. The controller triggers the behavior process to actually execute the body of the corresponding guarded method. During the operation of the behavior process, the controller signals the requesting client the state `WaitForCompletion`. When the method execution finishes, the controller signals the interface block the state `ReadReturnValue` (in the case of a non-void method). The interface

block then monitors read accesses to the client's return value address and signals the controller when all data has been read - the same way it monitors the parameter transfer. *

7.8.4 Scheduler

The scheduler's task is to select one of potentially multiple concurrent requests. Each client can issue only one request at a time, because a method call to a Shared Object is blocking and hence a client is not able to call any other method before the initially requested method has finished.

The hardware implementation of the scheduler is automatically derived from the given user-defined scheduler class. More specifically, the synthesis process extracts the schedule method of the given scheduler class and creates a process from it, which performs the given function. Additionally, the incoming requests are masked by the guards such that requests whose guard is `false` are not considered by the scheduler.

Compared to the scheduler synthesis as described in [135, 142, 97], the only modification is the additional `Schedule` signal. It is necessary, because the guards are not valid in every cycle and the scheduler must not start before the guards are up to date. Furthermore, an explicit start signal allows for an extension of a pure combinatorial scheduler to a multi-cycle scheduler.

7.8.5 Guard Evaluator

Because the state is stored in a RAM it is no longer possible to implement the guard expressions as combinatorial logic. Instead, the guard expression must be a single method call, e.g. `isNotFull()` in case of a FIFO's put method. Since it is not possible to access all data members, i.e. the whole object state, in parallel, it is consequently not possible to evaluate the guards in parallel - in the general case. Hence, the synthesized hardware implementation of the guard evaluator is similar to the behavior process: all guard methods (as opposed to the guarded methods) of the Shared Object are combined into one process. This process can be triggered by the controller to update some or all guards. The guard methods' results are hold in registers at the output of the guard evaluator and are fed into the scheduler.

7.8.6 Behavior Process

The behavior process finally implements the user-defined behavior given by the bodies of the guarded methods. The main difference to the synthesis as in [135, 142, 97] is the access to the arguments and the object's state, which both are stored in RAM as opposed to registers. There are two consequences of this approach. Firstly, special control signals for the RAM have to be generated and potentially more cycles are need for method's execution, especially when a member or argument is distributed over multiple memory words. Secondly, the memory layout of the argument RAM is crucial for the correct communication between client and behavior process. The memory layout of the state RAM, however, can be chosen freely during the synthesis, because it can only be accessed from within the Shared Object.

7.8.7 Potential Extensions and Optimizations

There are several extensions, which could be made to the synthesis strategy. Currently, it is not clear to which extend these extensions and optimizations could be realized in the synthesis tool prototype. These extensions/optimizations are:

- Block-Ram Parity: The Xilinx Block-Ram offers 16384 data bits and 2048 additional parity bits. It would be possible to use the extra parity bits to store data, too. The main difficulty is the fact, that the data bitwidths are no longer powers of 2 (9, 18 and 36 instead of 8, 16 and 32).

- Block-Ram Port Configuration: Each port of a Block-Ram can be configured independently. This could be used to optimize the data paths to certain client at the expense of a more complex addressing scheme: When the two ports are configured with different data bus

widths, their address bus widths are different, too, because both ports "see" the same number of bits in different word sizes.

- Mapping data members to state RAM: There are many ways of mapping data members to the state RAM. First, multiple data members could be packed into one word if the word size is larger than the member. The advantage is a better/maximum memory efficiency, the disadvantage is that write accesses to data member must be preceded by read accesses of the target memory address in order not to destroy the other attributes stored at the same memory word. Secondly, if different data members are not packed into the same memory word, the selection of the data word size determines the memory efficiency. For example, mapping combinations of 3, 16 and 33 Bit data members to different data bus widths results in different numbers of wasted bits.
- Combining argument and state RAM. If there are only few arguments and few data members, the Block Rams could be shared at the expense of additional multiplexers.
- Allocating client-specific argument and return value ranges in the argument RAM. If the Block-Ram is large enough to hold each client's arguments then the argument RAM could be statically partitioned during synthesis such that each client has a dedicated address range in the argument RAM. The advantage of such an approach is that the controller could already instruct the next client to send its parameters even though the behavior process is serving another request. Hence, the parallelism is increased and the delay of a guarded method invocation is reduced. For this approach to work it is necessary that the controller not only passes the method ID to the behavior, but also a start address at which the arguments of the granted client can be found.
- Distributing the argument RAM. The argument RAM could be distributed such that each client has its own argument RAM. In this case, each interface block would have its own dedicated Block-Ram port, which could be configured independent of all other interface block. Although this approach would save the ArgMux, it would require a multiplexer for the behavior process in order to access the different argument RAMs.
- Direct state RAM access. For simple buffer applications of a Shared Object, a direct access to the state RAM could decrease the latency of a guarded method. For example, a put method could directly write the item to the state RAM instead of writing it to the argument RAM and letting the behavior process copy the item from the argument RAM to the state RAM.

7.8.8 Hardware Client

In this section we will introduce the hardware interface synthesis of a hardware client to a Shared Object. A client is always a master and the server, i.e. the Shared Object is always the slave.

On application layer, a hardware client accesses a Shared Object via method calls on an `osss_port<osss_shared_if<...> >`. Such a communication link between the client and the Shared Object is mapped onto an OSSS Channel on the architecture layer. This mapping guides the hardware interface synthesis process. Two transactors are involved in the communication, namely the RMI transactor and the channel's master transactor. The master transactor determines the hardware client's signal-level port interface as well as the necessary protocol to drive these ports.

For the synthesis a predefined macro that represents combinations of the RMI- and the master transactor is used. More specifically, port-method-calls within the client's process are replaced by RMI-over-channel protocol.

```
1  template< unsigned int A, unsigned int B>
2  struct max_iter { enum { value = (A / B) + ((A % B) ? 1 : 0) }; };
3
4  template< unsigned int A, unsigned int B>
5  struct div { enum { value = (A / B) }; };
6
7  template< unsigned int A, unsigned int B>
8  struct mod { enum { value = (A % B) }; };
```

```
9
10 template<unsigned int N, unsigned int P=0>
11 struct Log2 { enum { value = Log2<N/2,P+1>::value }; };
12
13 template <unsigned int P>
14 struct Log2<0, P> { enum { value = P+1}; };
15
16 template <unsigned int P>
17 struct Log2<1, P> { enum { value = P+1}; };
```

Listing 7.8: Helper templates for compile-time calculations

Listing 7.8 shows some helper templates used for compile-time calculation of

- `max_iter<A, B>::value` $= \lceil \frac{A}{B} \rceil$
- `div<A, B>::value` $= \lfloor \frac{A}{B} \rfloor$
- `mod<A, B>::value` $= A \mod B$
- `Log2<N>::value` $= \frac{\log N}{\log 2} + 1$

```
1  SC_MODULE(P2P_Client) {
2    sc_in<bool> clock;
3    sc_in<bool> reset;
4
5    // Point-to-point channel signal interface, replaces
6    // osss_port<osss_shared_if< ... > > p;
7    sc_out<bool> client_strobe;
8    sc_out<sc_bv<N> > client_data;
9    sc_in<bool> server_strobe;
10   sc_in<sc_bv<M> > server_data;
11
12   void main() {
13     client_strobe = false;
14     client_data = '0';
15     ...
16     wait();
17     while(true) {
18       ...
19       // parameters for method call on port p:
20       // 1) method ID (MID_value) of called method
21       sc_bv<MID_size> MID =
22         static_cast<sc_bv<MID_size> >(sc_biguint<MID_size>(MID_value));
23       // 2) serialized method argument vector of size ARG_size_MID (in bit)
24       sc_bv<ARG_size_MID> argument_vector;
25       // 3) serialized return argument vector of size RET_size_MID (in bit)
26       sc_bv<RET_size_MID> return_argument_vector;
27
28       // request method call be sending Method ID (MID) to client
29       for(sc_biguint<Log2<max_iter<MID_size, N>::value> > i = 0;
30           i < div<MID_size, N>::value - 1; i++) {
31         client_strobe.write(true);
32         client_data = MID.range(i*N+N-1, i*N);
33         wait();
34       }
35       if (mod<MID_size, N>::value != 0) {
36         client_data =
37           MID.range(MID_size - 1, MID_size - mod<MID_size, N> - 1);
38         wait();
39       }
40
41       client_strobe.write(false);
42       wait();
43
44       // wait for grant
45       while(!server_strobe.read()) {
46         wait();
47       }
48       while(server_strobe.read()) {
```

```
49          wait();
50        }
51
52        // send/stream arguments
53        if(has_argument_MID) {
54          for(sc_biguint<Log2<max_iter<ARG_size_MID, N>::value> > i = 0;
55              i < div<ARG_size_MID, N>::value - 1; i++) {
56            client_strobe.write(true);
57            client_data = argument_vector.range(i*N+N-1, i*N);
58            wait();
59          }
60          if (mod<ARG_size_MID, N>::value != 0) {
61            client_data =
62              argument_vector.range(ARG_size_MID - 1,
63                ARG_size_MID - mod<ARG_size_MID, N> - 1);
64            wait();
65          }
66          // argument streaming completed
67          client_strobe.write(false);
68          wait();
69        }
70
71        // wait for completion
72        while(!server_strobe.read()) {
73          wait();
74        }
75        sc_biguint<Log2<max_iter<RET_size_MID, M>::value> > i = 0;
76        while(server_strobe.read()) {
77          // read/stream return argument
78          if(has_return_argument_MID) {
79            if(i < max_iter<RET_size_MID, M>::value) {
80              return_argument_vector.range(i*M+M-1, i*M) = server_data.read();
81              i++;
82            }
83          }
84          else
85            wait();
86        }
87        ...
88      }
89    }
90
91    SC_CTOR(P2P_Client) {
92      SC_CTHREAD(main, clock.pos());
93      reset_signal_is(reset, true);
94    }
95  };
```

Listing 7.9: Synthesizable template for a method call over a point-to-point channel

In Listing 7.9 the synthesizable template for a method call from a hardware client to a Shared Object using the simple point-to-point channel (see Definition 5.6.2.15) is shown. Here, the read-write master interface is implemented. The RMI protocol is realized on top of the read-write master interface and implements all RMI protocol phases: request method call, wait for grant, send/stream arguments, wait from completion, read/stream return argument.

```
1   SC_MODULE(IPIF_Client) {
2     sc_in<bool> clock;
3     sc_in<bool> reset;
4
5     // Bus IP (IPIF) signal interface, replaces
6     // osss_port<osss_shared_if< ... > > p;
7
8     // IP Master Request
9     sc_out<sc_bv<C_IPIF_AWIDTH> > IP2Bus_Addr;
10    sc_out<sc_bv<C_IPIF_DWIDTH/8> > IP2Bus_MstBE;
11    sc_out<sc_bv<C_IPIF_AWIDTH> > IP2IP_Addr;
12    sc_out<bool> IP2Bus_MstWrReq;
13    sc_out<bool> IP2Bus_MstRdReq;
14    sc_out<bool> IP2Bus_MstBurst;
15    sc_out<sc_bv<C_IPIF_MSTNUM_WIDTH> > IP2Bus_MstNum;
```

```
16
17  // Status Reply to IP Master
18  sc_in<bool> Bus2IP_MstWrAck;
19  sc_in<bool> Bus2IP_MstRdAck;
20  sc_in<bool> Bus2IP_MstLastAck;
21  sc_in<bool> Bus2IP_IPMstTrans;
22
23  // Slave Bus Interface
24  sc_in<sc_bv<C_IPIF_AWIDTH> > Bus2IP_Addr;
25  sc_in<sc_bv<C_IPIF_DWIDTH/8> > Bus2IP_BE;
26  sc_in<bool> Bus2IP_RNW;
27  sc_in<bool> Bus2IP_Burst;
28  sc_in<sc_bv<C_ARD_ID_ARRAY_length> > Bus2IP_CS;
29  sc_in<sc_bv<C_ARD_NUM_CE_ARRAY_length> > Bus2IP_CE;
30  sc_in<sc_bv<C_ARD_NUM_CE_ARRAY_length> > Bus2IP_RdCE;
31  sc_in<sc_bv<C_ARD_NUM_CE_ARRAY_length> > Bus2IP_WrCE;
32  sc_in<bool> Bus2IP_RdReq;
33  sc_in<bool> Bus2IP_WrReq;
34  sc_in<sc_bv<C_IPIF_DWIDTH> > Bus2IP_Data;
35
36  sc_out<sc_bv<C_IPIF_DWIDTH> > IP2Bus_Data;
37  sc_out<bool> IP2Bus_WrAck;
38  sc_out<bool> IP2Bus_RdAck;
39
40  void main() {
41    IP2Bus_Addr = '0';
42    IP2Bus_MstBE = '0';
43    IP2IP_Addr = '0';
44    IP2Bus_MstWrReq = false;
45    IP2Bus_MstRdReq = false;
46    IP2Bus_MstBurst = false;
47    IP2Bus_MstNum = '0';
48
49    IP2Bus_Data = '0';
50    IP2Bus_WrAck = false;
51    IP2Bus_RdAck = false;
52    ...
53    wait();
54    while(true) {
55      ...
56
57      // ID of this client
58      sc_unit<16> client_id = CID;
59
60      // parameters for method call on port p:
61      // 0) object ID (OID_value) of called remote object
62      sc_bv<OID_size> OID =
63        static_cast<sc_bv<OID_size> >(sc_biguint<OID_size>(OID_value));
64      // 1) method ID (MID_value) of called method
65      sc_bv<MID_size> MID =
66        static_cast<sc_bv<MID_size> >(sc_biguint<MID_size>(MID_value));
67
68      // request method call be sending Method ID (MID) to client
69      IP2Bus_MstRdReq = false;
70      IP2Bus_MstWrReq = true;
71      IP2Bus_MstBurst = false;
72      IP2Bus_MstNum = static_cast<sc_bv<C_IPIF_MSTNUM_WIDTH> >(1u);
73      IP2Bus_Addr =
74        static_cast<sc_bv<C_IPIF_AWIDTH> >(client_id_to_address(OID, CID));
75      IP2Bus_MstBE = '1';
76      IP2IP_Addr = static_cast<sc_bv<C_IPIF_AWIDTH> >(0u);
77      wait();
78
79      // wait for completion of master write transfer
80      while(!Bus2IP_MstLastAck.read())
81        wait();
82      while(Bus2IP_MstLastAck.read()) {
83        IP2Bus_MstWrReq = false;
84        wait();
85      }
86
```

```
87        // read Shared Object's message register periodically until call is granted
88        while(true) {
89          IP2Bus_MstRdReq = true;
90          IP2Bus_MstBurst = false;
91          IP2Bus_MstNum = static_cast<sc_bv<C_IPIF_MSTNUM_WIDTH> >(1u);
92          IP2Bus_Addr =
93            static_cast<sc_bv<C_IPIF_AWIDTH> >(object_id_to_address(OID));
94          IP2Bus_MstBE = '1';
95          IP2IP_Addr = static_cast<sc_bv<C_IPIF_AWIDTH> >(0u);
96          wait();
97          // wait for completion of master read transfer
98          while(!Bus2IP_MstLastAck.read())
99            wait();
100         while(Bus2IP_MstLastAck.read()) {
101           IP2Bus_MstRdReq = false;
102           wait();
103         }
104         if (stauts == GRANTED) break;
105       }
106
107       // stream (write) arguments
108       if(has_argument_MID) {
109         IP2Bus_MstWrReq = true;
110         IP2Bus_MstBurst = true;
111         IP2Bus_MstNum =
112           static_cast<sc_bv<C_IPIF_MSTNUM_WIDTH> >(
113             sc_biguint<C_IPIF_MSTNUM_WIDTH>(
114               max_iter<ARG_size_MID, C_IPIF_DWIDTH>::value));
115         IP2Bus_Addr =
116           static_cast<sc_bv<C_IPIF_AWIDTH> >(object_id_to_arguments(OID));
117         IP2Bus_MstBE = '1';
118         IP2IP_Addr = static_cast<sc_bv<C_IPIF_AWIDTH> >(LOCAL_ARG_BASE_ADDR);
119         wait();
120
121         // wait for completion of master write transfer
122         while(!Bus2IP_MstLastAck.read())
123           wait();
124         while(Bus2IP_MstLastAck.read()) {
125           IP2Bus_MstWrReq = false;
126           IP2Bus_MstBurst = false;
127           wait();
128         }
129       }
130
131       // read Shared Object's message register periodically until call is completed
132       // and return arguments are ready (if any)
133       while(true) {
134         IP2Bus_MstRdReq = true;
135         IP2Bus_MstBurst = false;
136         IP2Bus_MstNum = static_cast<sc_bv<C_IPIF_MSTNUM_WIDTH> >(1u);
137         IP2Bus_Addr =
138           static_cast<sc_bv<C_IPIF_AWIDTH> >(object_id_to_address(OID));
139         IP2Bus_MstBE = '1';
140         IP2IP_Addr = static_cast<sc_bv<C_IPIF_AWIDTH> >(0u);
141         wait();
142         // wait for completion of master read transfer
143         while(!Bus2IP_MstLastAck.read())
144           wait();
145         while(Bus2IP_MstLastAck.read()) {
146           IP2Bus_MstRdReq = false;
147           wait();
148         }
149         if (stauts == RETURN_READY) break;
150       }
151
152       // stream (read) return arguments
153       if(has_return_argument_MID) {
154         IP2Bus_MstRdReq = true;
155         IP2Bus_MstBurst = true;
156         IP2Bus_MstNum =
157           static_cast<sc_bv<C_IPIF_MSTNUM_WIDTH> >(
```

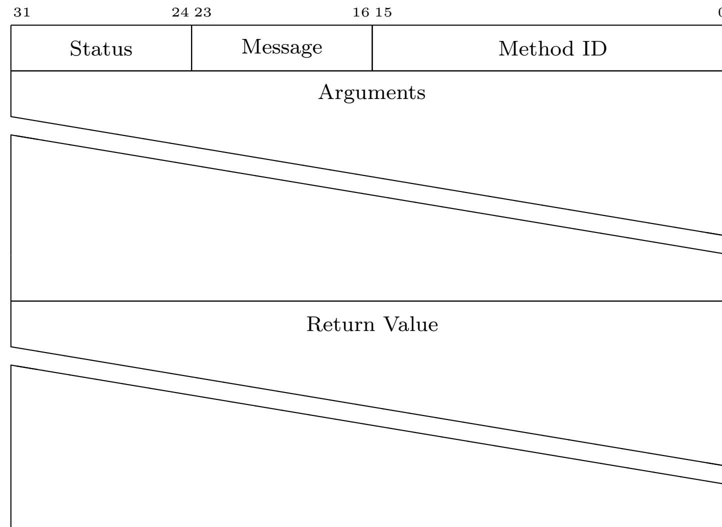


Figure 7.26: Memory layout (big-endian) of an RMI client for a 32 bit bus data width

```
158             sc_biguint<C_IPIF_MSTNUM_WIDTH>(
159               max_iter<RET_size_MID, C_IPIF_DWIDTH>::value));
160         IP2Bus_Addr =
161           static_cast<sc_bv<C_IPIF_AWIDTH> >(object_id_to_return(OID));
162         IP2Bus_MstBE = '1';
163         IP2IP_Addr = static_cast<sc_bv<C_IPIF_AWIDTH> >(LOCAL_RET_BASE_ADDR);
164         wait();
165 
166         // wait for completion of master write transfer
167         while(!Bus2IP_MstLastAck.read())
168           wait();
169         while(Bus2IP_MstLastAck.read()) {
170           IP2Bus_MstRdReq = false;
171           IP2Bus_MstBurst = false;
172           wait();
173         }
174       }
175       ...
176     }
177   }
178 
179   SC_CTOR(IPIF_Client) {
180     SC_CTHREAD(main, clock.pos());
181     reset_signal_is(reset, true);
182   }
183 };
```

Listing 7.10: Synthesizable template for a method call over a bus channel using IPIF

Listing 7.10 shows the synthesizable template for a method call from a hardware client to a Shared Object using a shared bus. For separating the physical and the media access layer of the used bus IP from the hardware client interface, the IPIF (see Section E.3.7) is used. For realizing RMI with the IPIF master interface, the method and return arguments need to be stored (in the serialized format) inside a memory of the same bitwidth as the connected IP channel. Figure 7.26 shows the memory layout (for a 32 bit bus) used inside hardware clients. The connection of the memory to the IPIF internal slave interface is not shown in Listing 7.10. For more information about the IPIF master interface usage see [108, 112].

7.9 Back-End Synthesis

Figure 7.27 shows the synthesis flow for the supported target platform. The upper part shows the tools and libraries developed for processing the OSSS design description while the lower part implements the back-end flow using Xilinx synthesis tools.

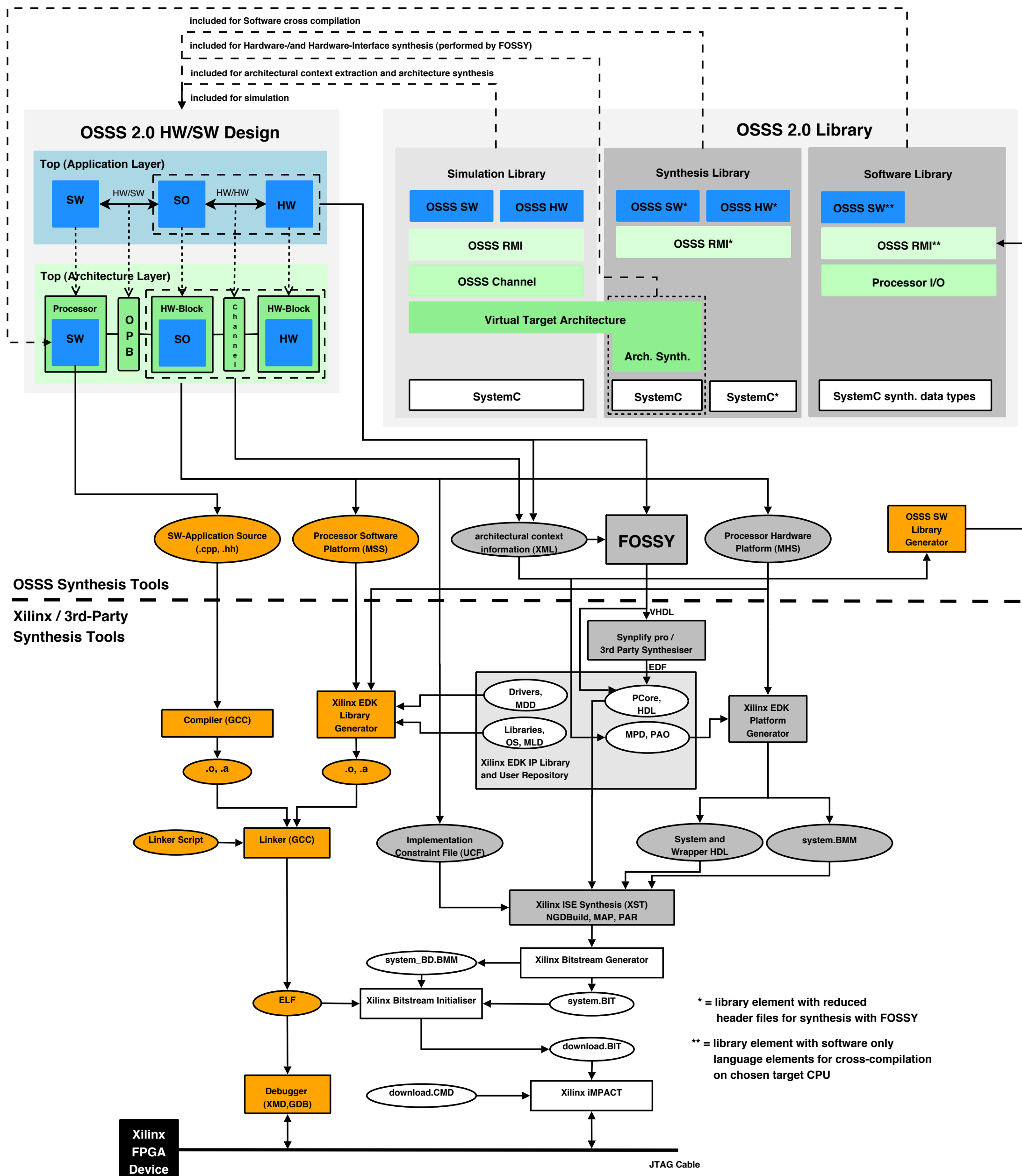


Figure 7.27: Overview of the OSSS Back-End Synthesis Flow [44]

The back-end synthesis flow considered here consists of the Xilinx EDK and the Xilinx ISE to implement an OSSS design on the targeted FPGA prototyping boards.

After running the Xilinx EDK Platform Generator, state-of-the art synthesis tools can be used to transform the synthesizable system description to a netlist. In Figure 7.27 the Xilinx Synthesis Technology (XST) synthesizer is used but is it possible to use any other 3rd party HDL synthesizer (e.g. Synplicity) that is capable of generating EDIF netlists. After the design has been synthesized it has to be implemented on the chosen FPGA. This step will be performed by the Xilinx Place & Route tool (PAR). Besides the synthesized netlist the Place & Route tool needs a constraint file (UCF file) which contains information about the floorplanning and the connections to the external pins. At the end of the ISE flow, a bitstream (system.BIT) that contains the configuration for the FPGA is generated.

7.9.1 Integration into Xilinx Flow

Xilinx provides the Integrated Software Environment (ISE), which can be used to synthesize, to simulate, to analyze and to download a hardware design to any Xilinx FPGA. As an input the Xilinx ISE takes VHDL, Verilog and/or EDIF netlists. By providing a constraint file (ucf file) the ISE can be used to synthesize a hardware design for a specific Xilinx FPGA device, to generate a configuration bitstream and to download it to the configuration memory of the FPGA.

The Xilinx Platform Studio (XPS) builds on top of the Xilinx ISE and can be regarded as a possible design entry of the ISE design flow. The aim of the XPS is the integration of various IP components including embedded processors, DSP blocks, peripherals and communication IPs. The XPS itself includes a library of several IP components including the MicroBlaze embedded soft-core processor plus several peripherals, which can be interconnected by using the IBM CoreConnect technology. Since the MicroBlaze is a soft processor core it can be fully customized by the designer instantiating it. By using the XPS a designer can assemble an architecture consisting out of different IP components including user defined hardware blocks. After assembling and configuration of the desired architecture the Xilinx ISE is used to synthesize, simulate and download the whole design to an FPGA.

7.9.1.1 Integrated Software Environment (ISE)

Figure 7.28 gives an overview of the design flow supported by the Xilinx ISE. The design entry is usually performed by providing VHDL, Verilog and/or EDIF netlists together with a constraint file. In our aspired design flow, we will use the Xilinx Platform Studio (XPS) to build the processor system and the overall system architecture. I.e. the XPS will be considered as the design entry for the ISE.

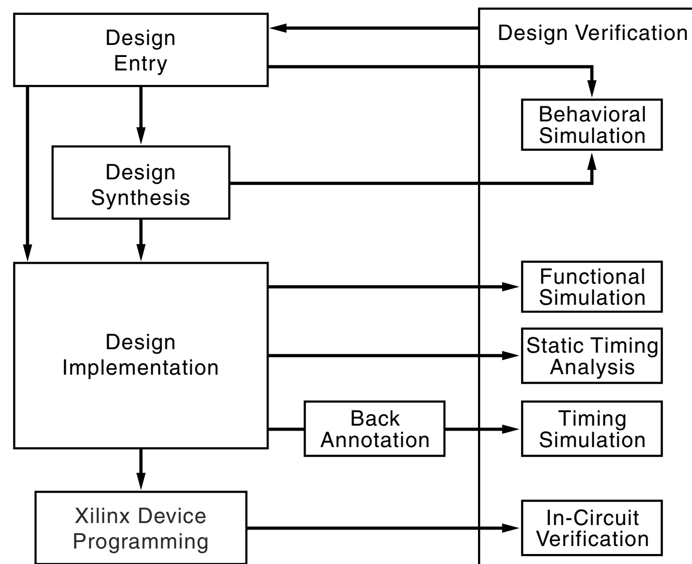


Figure 7.28: Design flow supported by the Xilinx ISE [118]

After the design entry has been performed the Xilinx Synthesis Technology (XST) or another third party HDL synthesizer can be used. The resulting netlist can be simulated either by external simulators like ModelSim or by the Xilinx Simulator integrated in the ISE. After the design has been synthesized it has to be implemented on the chosen FPGA. This step will be performed by the Xilinx place and route tool (PAR). Besides the synthesized netlist the place and route tool needs a constraint file (ucf file) which contains information about the floorplanning and the connections to the external pins. This placed and routed design can be simulated as before but with respect to the physical implementation on the chosen FPGA. Thus, this simulation is called timing simulation since it reflects a much more realistic timing than

the simulation after synthesis. Additionally, a timing analysis will be performed which can be used to check the fulfillment of external timing constraints or timing constraints specified in the constraint file. When all functional and timing requirements have been verified successfully, a configuration bitstream can be generated. The ISE also provides a programming tool (iMPACT) to download the configuration bitstream to the specific Xilinx device.

7.9.1.2 Xilinx Platform Studio (XPS)

As already stated above, the XPS builds on top of the ISE. In this section, we will roughly outline the design flow of the XPS. It mainly includes the following phases:

1. Hardware platform creation
2. Hardware platform verification using simulation
3. Software platform creation
4. Software application creation
5. Software verification using debugging

Hardware Platform Creation & Simulation Figure 7.29 shows the hardware platform creation flow using the XPS.

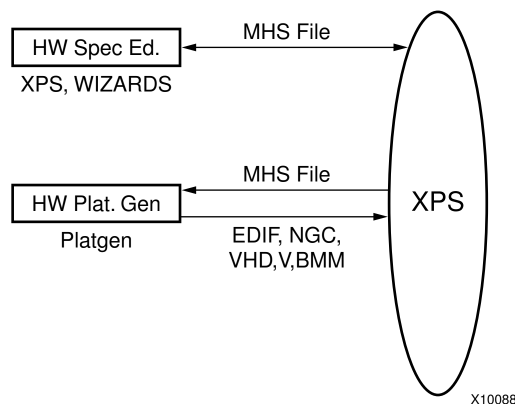


Figure 7.29: Hardware platform creation [95]

The hardware platform is defined by the Microprocessor Hardware Specification (MHS) file. The hardware platform consists of one or more processors and peripherals connected to the processor busses. Several useful peripherals (which will be considered as IPs) are supplied by Xilinx, along with the XPS tools. But you can define your own peripherals and include them in the MHS. The Microprocessor Hardware Specification file is a simple text file that can be created with any text editor.

The MHS file defines the system architecture consisting of peripherals, and embedded processors. It also defines the connectivity of the system, the address map of each peripheral in the system, and the configurable options for each peripheral. You can also specify multiple processor instances connected to one or more peripherals through one or more buses and bridges in the MHS.

The Platform Generator tool (Platgen) creates the hardware platform using the MHS file as input. Platgen creates netlist files in various formats such as EDIF and Xilinx proprietary NGC, support files for downstream tools, and top level HDL wrappers used to add custom designed components (synthesized by *Fossy*) to the automatically generated hardware platform.

After running Platgen, FPGA implementation tools, which are part of ISE, run automatically to complete the implementation of the hardware. At the end of the ISE flow, a bitstream is generated to configure the FPGA.

XPS also supports the simulation of the hardware platform before using Platgen for FPGA implementation. The simulation platform is based on the hardware platform. Instead of the Platgen tool the Simgen tool processes the MHS file to create simulation files, such as VHDL, Verilog, or various compiled models, along with some command files for specific simulators supported by the tool. If the software application that runs on the hardware platform is available in executable format, it can initialize memories in the simulation platform.

Software Platform Creation The software platform is defined by the Microprocessor Software Specification (MSS) file. The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is a simple text file that can be created using any text editor.

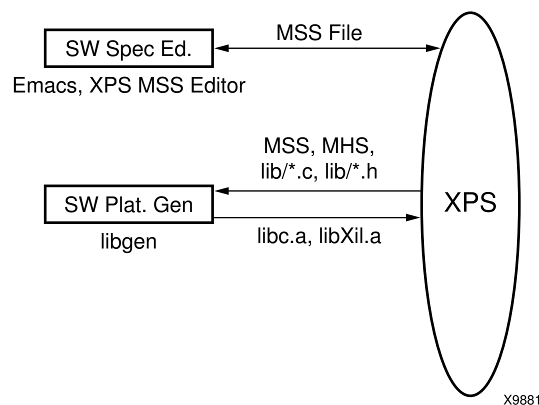


Figure 7.30: Software platform creation [95]

The MSS file is an input to the Library Generator tool (Libgen) for customization of drivers, libraries, and interrupt-handlers. The entire process of creating the software platform is shown in Figure 7.30.

Software Application Creation & Software Debugging The software application is the code that runs on the processor(s) defined by the hardware platform. The source code for the application can be written in a "high level" language such as C or C++. The created source files are compiles and linked to generate executable files in the Executable and Link Format (ELF). Therefore, GNU compiler tools for the MicroBlaze (Mb-gcc) and the PowerPC (ppc-gcc) are used. The Xilinx Microprocessor Debugger (XMD) and the GNU Debugger (GDB) are working together to debug the software application.

The Xilinx Microprocessor Debugger (XMD) provides an instruction set simulator, and is optionally connected to a working hardware platform to allow GDB to run the application. This entire process is depicted in Figure 7.31.

7.10 Summary

In general, the synthesis flow can be subdivided into an OSSS specific part and a 3rd party back-end flow. Since we have chosen a Xilinx FPGA with a Xilinx MicroBlaze soft processor core the back-end flow consists out of the state-of-the-art Xilinx synthesis tool chain. For more information concerning the Xilinx specific tools please refer to [239, 238].

As a precondition, we assumed that the design has been partitioned into hardware and software parts. The resulting hardware/software design is OSSS compliant and has been refined from the *Application* to the *Architecture Layer*. This refinement step includes the refinement of each software task, each Shared Object and each hardware module to a description that does not violate the OSSS synthesis subset. Besides this behavior refinement the mapping to a specific

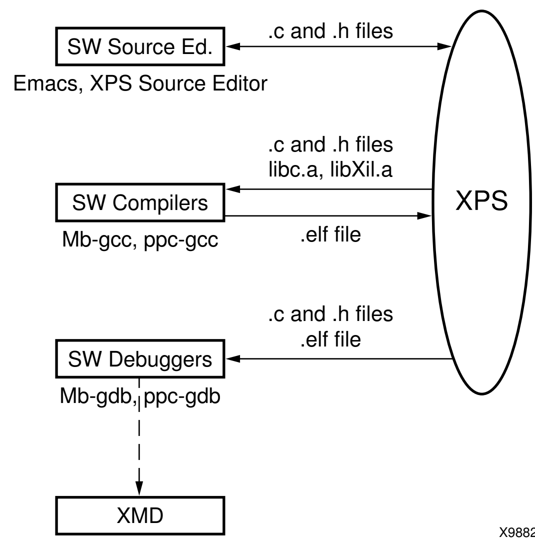


Figure 7.31: Software application creation & software debugging [95]

target architecture has been performed. It basically includes the mapping of a software task to a certain processor and the mapping of the abstract communication links from the *Application Layer* to an OSSS-Channel on the *Architecture Layer*.

To check the functionality and dynamic behavior of this hardware/software design on different levels of abstraction the OSSS simulation library is used in conjunction with the SystemC discrete event simulator. The OSSS simulation library provides classes for the modeling of hardware and software parts of the system on the *Application Layer*. Additionally, it provides several architecture elements that can be used to assemble the *Virtual Target Architecture*. This *Virtual Target Architecture Library* includes certain processors, memory blocks, sockets for Shared Objects, and OSSS-Channels (see Section 6.5). Besides the structural information provided by these building blocks, the OSSS-Channel enables a clock cycle accurate simulation of the communication on signal level.

After a successful simulation of the modeled hardware/software system the synthesis process can be started. The synthesis flow can be divided into the following phases:

1. OSSS synthesis tools
   (a) Architectural context extraction and hardware/software architecture synthesis (by using the OSSS Synthesis Library together with an architecture synthesis back-end for the *Virtual Target Architecture Library*)
   (b) Software library synthesis (by configuring the OSSS *Software Library* with information obtained during architectural context extraction)
   (c) High-level synthesis of the user-defined hardware part of the design (by using *Fossy* together with the OSSS *Synthesis Library* constituting a "header-only" version of the software, hardware, RMI and the SystemC library)
2. Xilinx synthesis tools
   (a) Platform generation using the Xilinx Embedded Development Kit (EDK)
   (b) RTL synthesis, mapping and place & route (this step can either be performed by the Xilinx Synthesiser Tools (XST) or by a third party tool supporting Xilinx devices like Synplify Pro [218])
   (c) Software library generation for the low-level software drivers for Xilinx specific IP cores by using the EDK (this includes the MicroBlaze soft processor core and the On-Chip Peripheral Bus, OPB)

(d) Cross-compilation and linking of the software part of the design by using the GNU compiler tool chain for the Xilinx MicroBlaze processor (part of the EDK)

(e) Bitstream initialization and downloading to the hardware platform

In the following Table 7.6 a review of all synthesis related goals from Chapter 2 is given.

Table 7.6: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled)

ID	Goal	Status	Comment
G1	Integration of synthesis tool and simulation infrastructure into Eclipse CDT Framework	●	Since OSSS is based on C++ and integration into the Eclipse C/C++ Development Tooling (CDT) Framework is possible. Both, the simulator and the synthesis tool *Fossy* have been successfully integrated, see Chapter G.
S1	Provide a (prototypical) synthesis tool	●	*Fossy* is a prototypical synthesis tool Shared Object and OSSS Hardware Module to VHDL synthesis, configuration and integration of these custom hardware blocks into an SoC architecture for Xilinx FPGAs.
S2	Software output language C++ compliant with C++ standard (ISO/IEC 14882:1998)	◖	In the current implementation *Fossy* does not generate any software code. The designer needs to write C++ code inside the Software Modules, which is compliant with ISO/IEC 14882:1998.
S3	Hardware language VHDL compliant with the synthesizable subsets of Synopsys Design Compiler and Synplify Pro from Synplicity	◖	The *Fossy* generated VHDL code has been successfully synthesized with Synplify Pro a Xilinx FPGA. The Synopsys Design Compiler for an ASIC target has not been tested so far.
S4	The generated code has to be readable for a human being	◖	The *Fossy* generated VHDL and SystemC code is human readable. All identifier names are preserved (some of them with pre-fixes). Type transformation (which is necessary to map the SystemC data type semantics to the VHDL semantics) introduces some additional casts. The state-machine transformation which translates implicit to explicit state-machines transforms `SC_CTHREADS` into `SC_MODULE` introducing explicit states states and next-state logic. With these transformations the code should stay human readable.
S5	Possibility to map the abstract communication objects onto concrete mechanisms such as memory mapped IO/ shared memory (using polling, interrupts and/or DMA) or proprietary direct HW/HW communication and to generate the necessary HW and SW parts	◖	OSSS Application Layer Communication Links can be mapped to a bus using memory mapped IO (currently only polling access is supported) or to any proprietary direct HW/HW communication. Interrupts and DMA are currently not supported.

continued on next page

Table 7.6: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
S6	For the integration of IP components it is necessary that the designer can control the synthesis and to enforce a certain communication mechanism, which is required by the IP component	●	The supported modeling style of Hardware modules is behavioral RTL which enables a clock-cycle accurate protocol description suitable for RTL IP component integration.
S7	Control of the synthesis by constraints in the synthesis script or by special statements within the source code	○	Currently not supported by *Fossy*.
S8	Efficiency of the generated code (for hardware: area and critical path; for software: memory footprint) compared to a hand-crafted design	◖	The efficiency of the generated custom hardware VHDL code in terms of area and critical path length, mainly depends on the SystemC input code. We have shown for an industrial use-case (see Section 8.4) that the *Fossy* generated VHDL code has an area overhead of 16% and a maximum clock frequency reduced by 3% compared to a hand optimized VHDL design.

CHAPTER 8

Experiments

8.1 Introduction

This chapter contains experiments for the demonstration and evaluation of the presented methodology (see Chapter 5), simulation (see Chapter 6) and synthesis (see Chapter 7). Table 8.1 gives an overview of the presented experiments, metrics, covered requirements and links to the respective section. This chapter closes with a discussion of meeting the goals from Chapter 2.

8.2 JPEG Encoder

8.2.1 Goals of this experiment

The focus of this experiment is on demonstrating feasibility and effectiveness of the OSSS Behavior Layer modeling elements. For evaluation of the OSSS Behavior Layer and comparison with SpecC, a JPEG encoder design [166, 165] has been ported from SpecC to OSSS Behavior.

The original model is described on four levels of abstraction (specification, architecture, communication and implementation) following the SpecC refinement methodology [160]. The specification, architecture and communication models and refinement steps between these models have been implemented in performed in OSSS. The different model's execution times will be compared between the OSSS Behavior and the SpecC reference design.

This experiment has been originally published in [43].

8.2.2 Introduction to JPEG

JPEG is a standard for image compression. It applies to either the full-color images or the gray-scale images of natural, real-world scenes. Today, parts of JPEG are available as software-only packages or together with specific hardware support. There are four modes of the operations in the JPEG standard: the sequential discrete cosine transform DCT-based mode, the progressive DCT-based mode, the lossless mode and the hierarchical mode. Our design employs the first mode, the sequential DCT-based mode, which is the simplest and the most commonly used mode.

Based on the sequential DCT-based mode, the JPEG encoder is divided into four functional blocks: the image fragmentation block, the DCT block, the quantization block and the entropy-coding block. The corresponding block diagram in Figure 8.1 illustrates the communication relationship between these blocks.

In the image fragmentation functional block, an image is divided into the non-overlapping data blocks, each of which contains an 8×8 matrix of pixels. In the DCT functional block, each data block is transformed into a frequency representation. There are two commonly used DCT

Design/-Experiment	Metrics	Covered Goals	Published	Section
JPEG Encoder	Evaluation of OSSS Behavior Layer modeling elements. Evaluation of hierarchical composition using Behavior diagrams. Comparison of simulation speed with SpecC reference implementation.	M1-M3, A2-A4	[43]	8.2
Adaptive Video Filter	Stepwise functional C++ to FPGA platform refinement (Behavior → Application → Virtual Target Architecture) with IP integration.	M1-M6, M8-M10, M14	[41]	8.3
NightView Video Filter	Shared Object for HW/SW communication. Application to Virtual Target Architecture mapping. Comparison with VHDL simulation speed and model complexity. Evaluation of synthesis results (area, critical path) against VHDL reference model.	M1-M6, M8-M10, M14, A2-A5, S1, S5, S6, S8	[62, 29]	8.4
MP3 Decoder	Evaluation of C++ and RMI overhead for HW/SW communication compared to optimized C-based memory mapped I/O communication.	M1-M10, A2, A3, S5, S8	[66]	8.5
IPv4 Packet Switch	Exploration of HW/SW partitioning and communication link to RMI/OSSS channel mapping. Feasibility of the Shared Object synthesis approach. Assessment of OSSS productivity gain: Shared Object synthesis compared to a manual SystemC primitive channel refinement.	M1-M9, A2-A3, S1, S5, S8	[65, 22]	8.6
JPEG 2000 Decoder	Exploration of the most promising parallel structure by comparing different HW/SW design alternatives (incl. multiple Software Tasks) on the Application Layer. Comparison of OSSS custom hardware synthesis with standard design approach using an industrial C++/VHDL-based FPGA implementation.	M1-M10, A2-A4, S1, S5, S8	[62, 45]	8.7

Table 8.1: Overview of experiments

algorithms for this translation process: the standard DCT and the ChenDCT. The ChenDCT algorithm is employed in our design. In the quantization block, the DCT output coefficients are quantized. Finally, in the entropy-coding block, the AC coefficients are encoded by using a predictive coder and the DC coefficients are encoded by using a run-length coder. Then the Huffman coding algorithm is employed to generate the JPEG image file.

The file I/O for the JPEG Encoder modeled in the testbench (as shown in Figure 8.2) is not part of the algorithm itself. First, the height (H) and width (W) of the image are passed from the bitmap input file reader to the JPEG Encoder, which uses them to determine the number

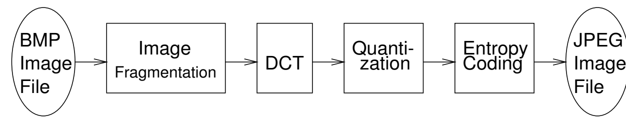


Figure 8.1: Block diagram of the JPEG encoder [165]

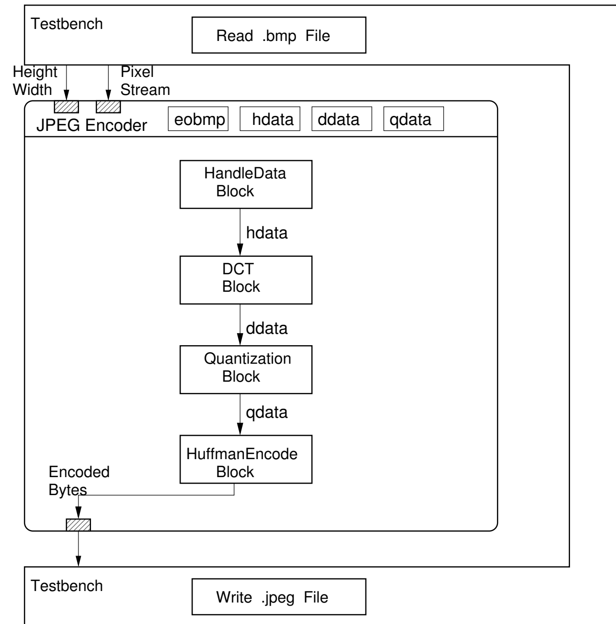


Figure 8.2: JPEG encoder model with testbench for file I/O [165]

of the iterations for the block transfer ($H * W$). Then serials of the image pixel streams (8-bit wide) are passed to the JPEG Encoder where the image is packed into blocks and then processed block by block. Each block consists of $8 * 8$ bytes of image information. The processed image is then sent out byte by byte to the file writer block where the final JPG image file is generated.

The JPEG encoder includes four basic blocks: HandleData, DCT, Quantization and HuffmanEncoder.

HandleData reads the inputs $H * W$ and pixel stream from the input file reader, calculates the number of the iterations, groups the pixel stream into $8 * 8$ pixel matrix (MCUs) and sends the MCUs to the DCT block.

DCT reads the MCUs passed in from the HandleData block, pre-shifts the MCUs, performs the Chen forward DCT algorithm on them, and sends the result (still $8 * 8$ matrix called transformed MCUs) to the Quantization block.

Quantization uses a quantization table to quantize each element of the resulting MCUs passed in from the DCT block and sends the result to the HuffmanEncoder block.

HuffmanEncoder performs a Huffman entropy-encoding and a run-length-encoding (RLE) on the successive bytes in the incoming MCUs. Each byte is transformed into a bit-sequence (often smaller than the 8 bits of the input byte). The sequence of encoded bits is then packed into bytes and written to the output file writer block.

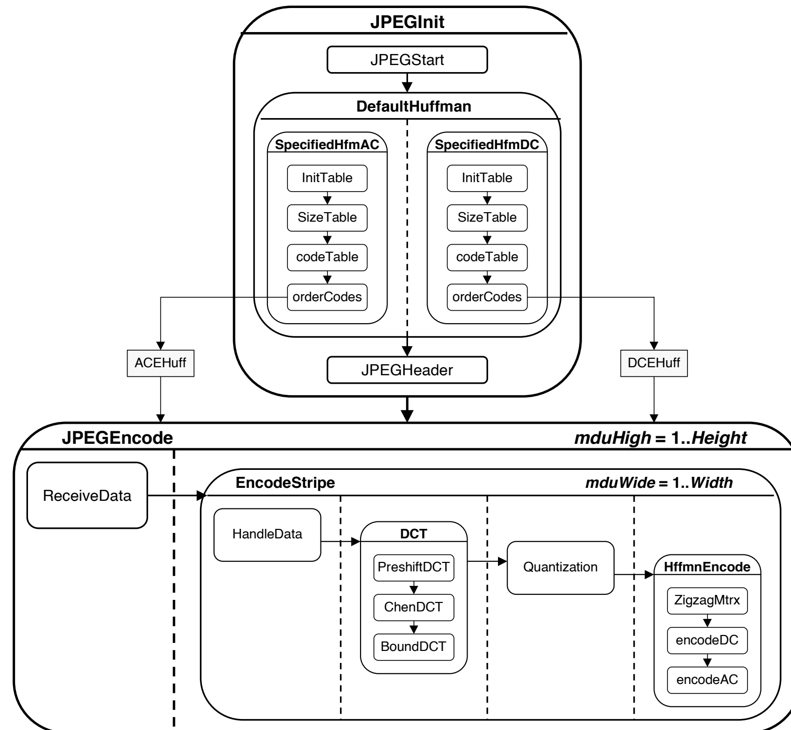


Figure 8.3: Structure of the JPEG encoder specification model [141]

8.2.3 JPEG encoder model

During this experiment, we are going to look on the first three models of a JPEG encoder: Specification, Architecture, and Communication Model. The fourth model, called *implementation model*, is not considered here:

Specification Model (*spec*) is untimed (or rather causal-timed) and exploits the parallelism available from the JPEG encoding algorithm, as shown in Figure 8.3. It consists of two sequential behaviors, `JPEGInit` followed by `JPEGEncode`. `JPEGInit` performs initialization of the two Huffman tables in two parallel sub-behaviors, and writes the output header. Then, the actual encoding is done in two nested, pipelined loops. The outer pipeline splits the image into stripes of 8 lines each. The inner pipeline then splits the stripes into 8 $\times$ 8 blocks and processes each block through DCT, quantization and Huffman encoding. As an example of communication, Figure 8.3 shows the two Huffman tables `ACEHuff` and `DCEHuff` that are sent from `JPEGInit` to `JPEGEncode`. Since these two behaviors are composed sequentially, channels can degenerate to simple variables.

Architecture Model (*arch*) is obtained after hardware/software partitioning and approximately timed (i.e. leaf-behaviors with annotated execution times), as shown in Figure 8.4. The Discrete Cosine Transform (DCT) is implemented in hardware while all other functionality is implemented in software. For the purpose of computation synthesis, we assumed a mapping of the encoder on an embedded processor (SW) assisted by a custom hardware co-processor (HW) for acceleration of the DCT. Software and hardware communicate via two message-passing channels, sending and receiving 8 $\times$ 8 blocks from software to the DCT processor and back. Behaviors inside the SW processor are statically scheduled and serialized. The two nested pipelines are converted into two nested, sequential loops. In Figure 8.4, the software waits for the result of the DCT before continuing with any processing. By changing only a few lines of code, you will be able to modify the architecture such that software and hardware operate in a pipelined fashion (i.e. while the DCT is processing a block the software continues processing of the previous block and prepares the next one), resulting in 100% utilization of the SW processor. Similarly, other architectural

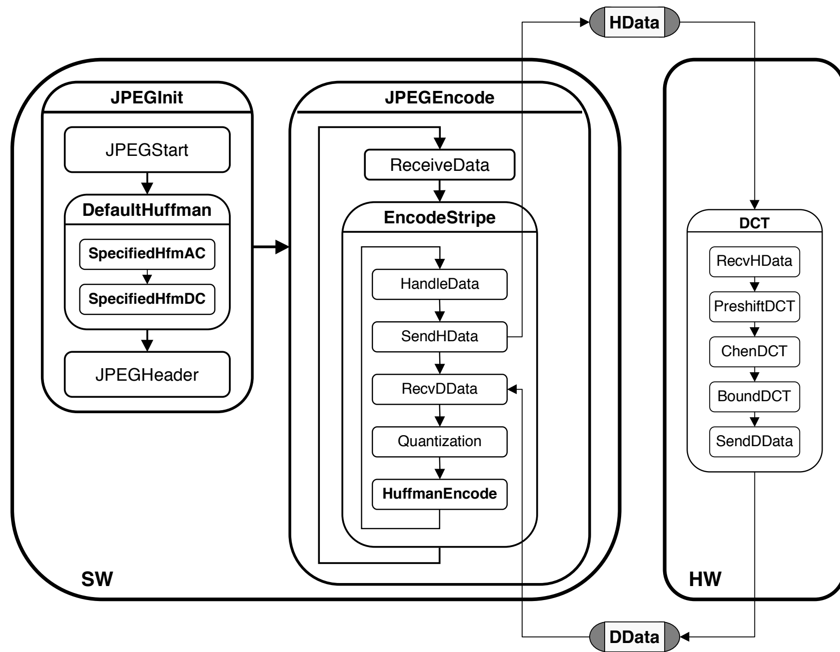


Figure 8.4: Structure of the JPEG encoder architecture model [141]

alternatives can be easily explored in a very short amount of time with minimal changes in the model.

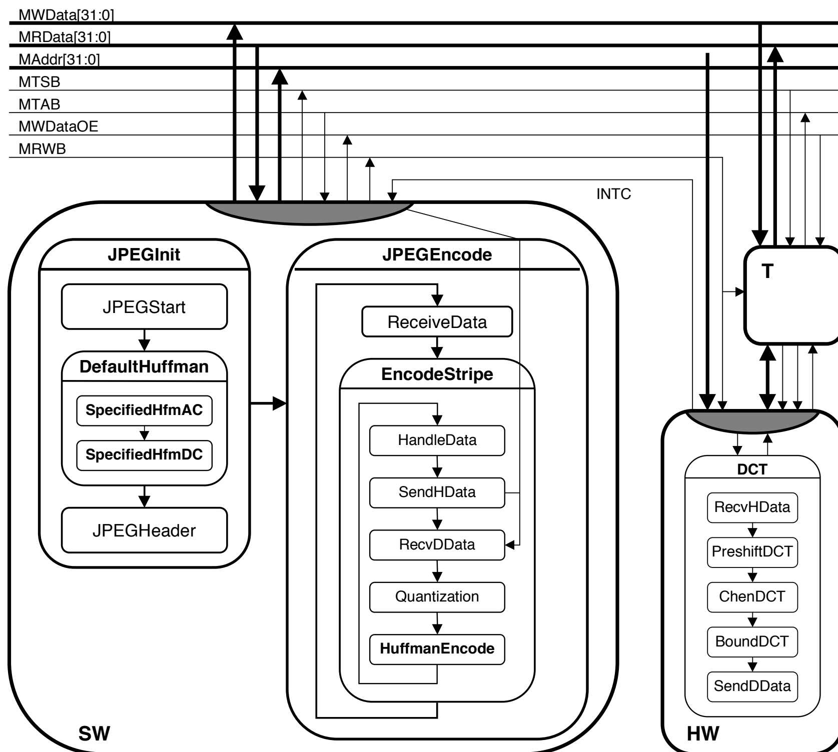


Figure 8.5: Structure of the JPEG encoder communication model [141]

Communication Model (*comm*) implements the hardware/software communication is imple-

mented by a cycle and bit accurate model of a bus, as shown in Figure 8.5. Finally, for communication synthesis, we connected the two processors via a single bus using a bit-level bus protocol. Furthermore, it was assumed that the protocol of the DCT IP is fixed and incompatible with the bus protocol, necessitating the inclusion of a transducer (Figure 8.5). The SW processor is the master on the bus and drives the address and control lines. The HW co-processor listens directly on the address bus and its associated control lines while the transducer translates between data transfer protocols. For synchronization, the hardware signals the software through the processor's interrupt line INTC. Inside the two PEs, bus drivers and interrupt handlers translate the message-passing calls of the behaviors into bus transactions by driving and sampling the PE's bus ports according to the protocol.

8.2.4 Results

The provided OSSS Behavior Layer modeling elements are capable of representing hierarchical SpecC designs using sequential (SEQ), finite-state machine (FSM, not shown in this experiment), parallel (PAR) and pipelined (PIPE) behavior composition. Communication between these hierarchical behaviors can be expressed via a set of pre-defined and user-defined hierarchical channels and signals. To demonstrate and evaluate the correct hierarchical behavior composition, including communication, OSSS Behavior models are capable to export their structural and behavioral composition and hierarchy during model elaboration phase. This exported data can be visualized using *Graphviz* [149], a graph visualization software. The chosen graphical representation is a follows:

Behaviors are represented as a rectangles with the name of the behavior. For composite and special leaf behaviors, the behavior type is given in square brackets, e.g. `m_main [PAR]`. Hierarchical behaviors are represented graphically by nested rectangles. A behavior's `main` routine is depicted as an ellipse (dashed line for non-leaf behaviors).

Shared Variables are represented as hexagons with the name of the shared variable. Piped variables have an extra attribute `[piped]`.

Channels are represented as octagons with the name of the channel. Hierarchical and pre-defined channels have extra attributes, such as `[hierarchical]`, `[osss_double_handshake_channel]` or `[sc_signal]`.

Execution sequence for sequential behaviors is illustrated through connecting the `main` routines by arrows in the execution order as defined in the behavior. The execution sequence of pipelined behaviors is illustrated through connecting the pipeline stage sub-behaviors by arrows.

Figure 8.6 shows an example graph structure of the model's `JPEGinit` behavior (compare with Figure 8.3).

The following structural representations have been extracted from the OSSS Behavior models:

- Figure H.1 represents the specification model as shown in Figure 8.3,
- Figure H.2 represents the architecture model as shown in Figure 8.4, and
- Figure H.3 represents the communication model as shown in Figure 8.5.

Table 8.2 and Figure 8.7 shows the different JPEG encoder model execution times measured for input bitmap images of four different dimensions. The results show that the OSSS Behavior implementation runs significantly faster than the SpecC reference implementation. This becomes most apparent when comparing the execution times of the *comm* models.

The main reason for this faster execution are:

1. the OSSS Behavior implementation only uses `SC_THREAD`s with dynamic sensitivity. Using `SC_CTHREAD`s with static sensitivity in combination with `wait()` or `wait(n)` calls reduces the simulation performance tremendously. Therefore, our implementation forbids using

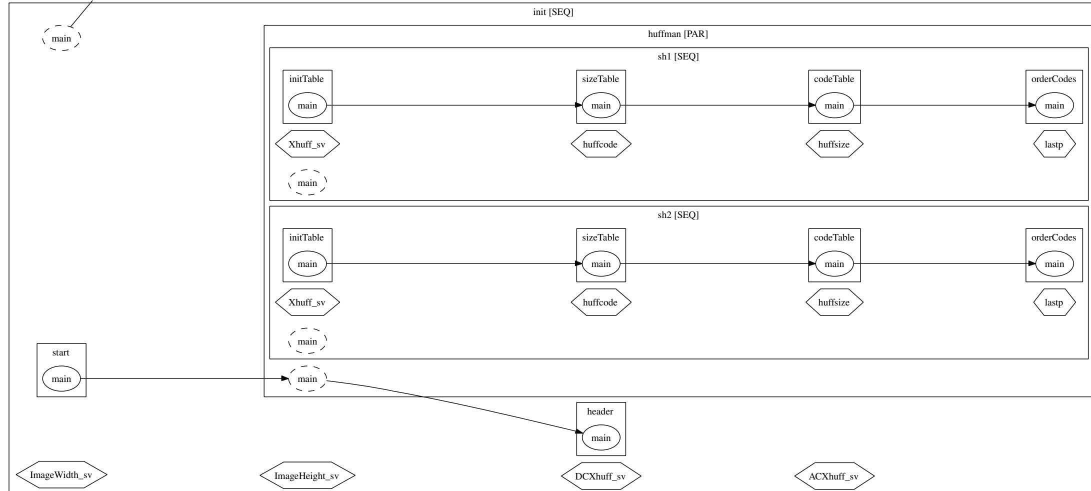


Figure 8.6: Example: Structure of the JPEG encoder specification model's `JPEGinit` behavior

model	image dimensions [pixel] 116 × 96	256 × 256	461 × 346	512 × 512
OSSS specification[a]	0.088 s	0.455 s	1.103 s	1.762 s
SpecC specification[b]	0.106 s	0.564 s	1.335 s	2.122 s
OSSS architecture[a]	0.056 s	0.265 s	0.640 s	0.997 s
SpecC architecture[b]	0.109 s	0.593 s	1.423 s	2.226 s
OSSS communication[a]	0.233 s	1.271 s	3.128 s	5.011 s
SpecC communication[b]	1.099 s	6.396 s	15.364 s	25.445 s

[a] with OSCI SystemC 2.2
[b] with SCRC 2.1, both on Intel® Core™2 CPU 6600@2.40GHz

Table 8.2: JPEG encoder model execution times

that kind of synchronization. Internally only spawning threads with annotated durations (i.e. clock periods) are used. Moreover, our implementation provides a `wait` function wrapper to enable convenient clock cycle timing annotation (as used in `SC_CTHREAD`s) in the communication model and Estimated Execution Time (EET) blocks in the architecture model.

2. the SystemC data types (i.e. integer types `sc_int<...>`, `sc_uint<...>`, `sc_bigint<...>`, `sc_biguint<...>` and `sc_bv<...>`) operate faster than the comparable data types in the non-commercial SpecC reference compiler. This becomes most apparent in the execution speed difference of the communication models where these data types are used inside the communication channels.

8.2.5 Conclusion

In this experiment a JPEG encoder specification, architecture and communication model have been implemented in OSSS. The SpecC reference implementation of the JPEG encoder has been compared with the OSSS implementation for functional equality through JPEG image output comparison. The hierarchical behavior composition and scheduling has been compared

1. statically though behavior hierarchy and scheduling graph generation and comparison

2. and dynamically through model execution and behavior trace comparison between the SpecC and OSSS JPEG encoder models.

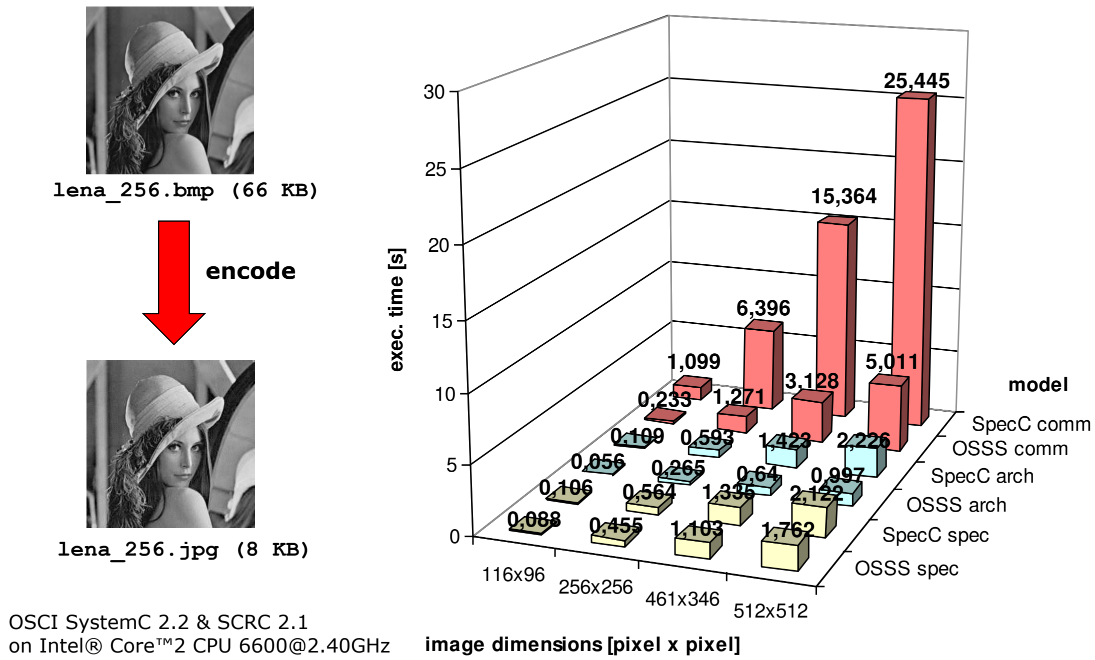


Figure 8.7: Comparison of JPEG encoder model execution times [43]

A comparison of the model execution times showed that the OSSS Behavior model implementation executes faster than the reference SpecC model using the SCRC 2.1 SpeC compiler.

	C	C++	VHDL	Verilog	SpecC	SystemC™	OSSS Behaviour	
Behavioral hierarchy	○	○	○	○	●	○	●	
Structural hierarchy	○	○	●	●	●	●	●	
Concurrency	○	○	●	●	●	●	●	
Synchronization	○	○	●	●	●	●	●	
Exception handling	◐	●	○	●	●	○	○	not implemented
Timing	○	○	●	●	●	●	●	
State transitions	○	○	○	○	●	○	◐	only TOC
Composite data types	●	●	●	◐	●	●	●	

Source: Rainer Dömer
University of California, Irvine

Figure 8.8: Comparison of OSSS Behaviour features

Figure 8.8 shows a comparison of some modeling features between SpecC, SystemC and OSSS Behavior. The `trap` and `interrupt` features (called exceptions) have not been implemented and thus not evaluated in OSSS. Regarding the description of state transition systems, OSSS only implements the Transition On Completion (TOC) feature of SpecC. For a technical discussion of these feature see Section 6.3. In conclusion, the modeling of finite-state machines in OSSS has not been evaluated in this design example.

For more information about implementation details, a proof of concept implementation of the OSSS Behavior Layer, including the JPEG encoder design example (and others), can be obtained from `http://system-synthesis.org/_media/osss-behaviour-0.0.2.tar.gz`.

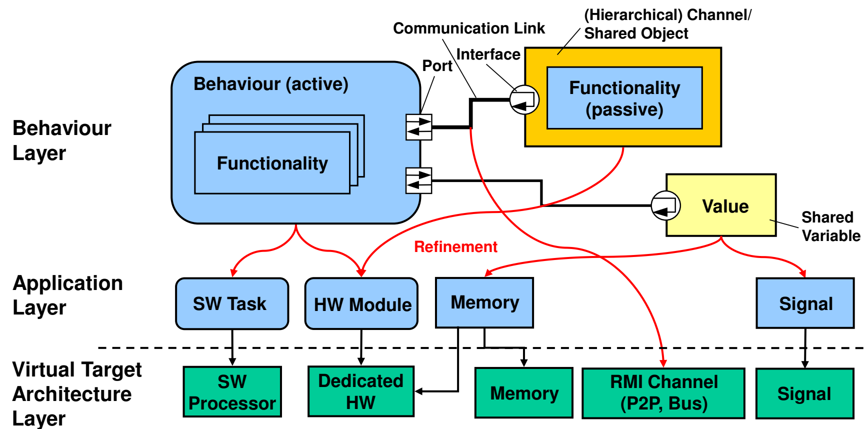


Figure 8.9: Model composition and refinement in OSSS [41]

8.3 Adaptive Video Filter

8.3.1 Goals of this experiment

The goal of this experiment is to demonstrate the stepwise refinement from a functional C++ description, to a parallel and pipelined OSSS Behavior Layer model, to a hardware/software partitioned OSSS Application Layer model, its mapping to an OSSS Virtual Target Architecture Layer model and finally, the synthesis and integration into an FPGA-based video processing platform.

This experiment demonstrates how generic functional C++ code, in this case

1. functors for data type independent calculations,

2. multi-dimensional arrays for data type independent storage,

can be used throughout the entire OSSS design flow, from the Behavioral to the Virtual Target Architecture Layer.

For optimizing and integrating the design into an FPGA video processing platform, RT level IP component integration of a memory block and signal level I/O refinement at the design boundaries will be performed.

This experiment has been originally published in [41].

8.3.2 Model Composition

OSSS Behavior Layer models can be composed of behaviors and channels (e.g. Shared Objects or Shared Variables) as shown in Figure 8.9. Behaviors follow the Program-State Machine semantics and can be hierarchically composed (cp. Section 6.3). They have their own thread of control and thus can be considered as *active*. They can be arranged in sequential, parallel or pipelined execution order. During hardware/software partitioning, behaviors are specialized to become either software tasks or hardware modules. This decision can be supported by different functional implementations of the same behavior. Shared Variables can either be implemented by signals or a dedicated memory. In this work Shared Objects always become dedicated Hardware. Communication links constituted by the port-interface-bindings are mapped onto OSSS Remote Method Invocation (RMI) channels. They serve as protocol wrappers around physical communication resources like point-to-point channels and buses.

8.3.3 Modeling in OSSS

In this section the OSSS methodology is applied to the implementation of an adaptive video filter. Its basic operation is a discrete 2D convolution

$$c'(x,y) = \sum_{i=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} \sum_{j=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} h(i,j) \cdot c(x-i, y-j)$$

with filter matrix size $N \in \{x \in \mathbb{N} \mid x \mod 2 \neq 0\}$, adaptive $N \times N$ filter kernel h, input image c, and filtered image c'. This filter operation should be applied to a video stream with a resolution of 720×480, 24 bit RGB, and a refresh rate of 60 Hz (DVD quality). For outputting this as a progressive picture (full image) a pixel clock of 27 MHz is required. This results in a data throughput of 648 MBit/s.

Figure 8.10 illustrates the OSSS top-down design flow for the adaptive video filter design. We start with an algorithmic specification of the filter chain as initial untimed OSSS Behavior functional model (see Figure 8.10a).

8.3.3.1 Behavior Layer Model

The Behavior Layer model of the video filter design in Figure 8.10a is used to explore the possible parallelism of a purely sequential C++ functional specification.

This involves determination of a valid scheduling and the explicit modeling of communication. All functional blocks from the sequential C++ specification model need to be wrapped by OSSS Behaviors, which can be arranged in sequential, parallel or pipelined execution order. The communication between behaviors is implemented using shared and piped variables for simple unidirectional communication. Double Handshake Channels (Video Stream Source and Video Stream Sink) are used for the synchronization and communication with the environment/test-bench.

In a first step, we reduce the RGB color space to luminance (gray scale) by applying the weighted sum

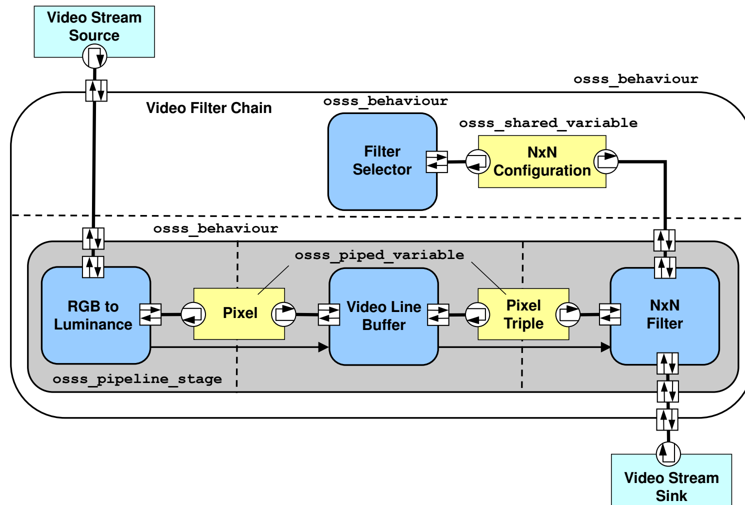
$$\text{RGB}_{\text{luminance}} = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B$$

to each pixel from the video stream source. This function is implemented as functor, shown in Listing 8.1. To enable reuse the functor is configurable for different color model bit widths.

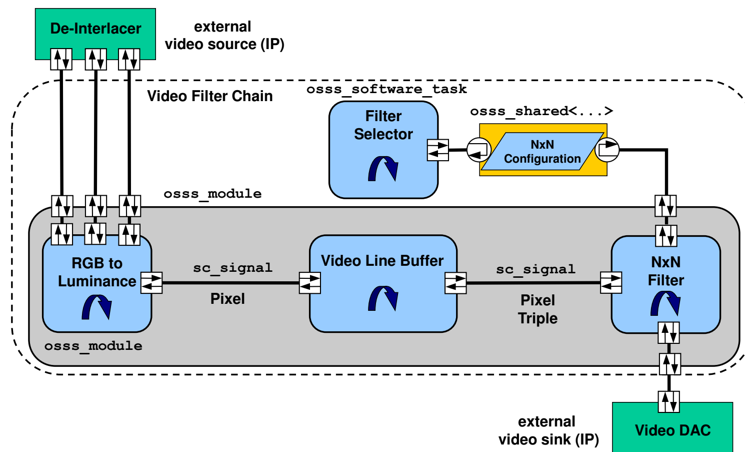
```
1  template <typename ColorChannelValue, typename GrayChannelValue>
2  struct RGB_To_Luminance {
3
4    GrayChannelValue
5    operator()(ColorChannelValue r, ColorChannelValue g, ColorChannelValue b) const {
6      return static_cast<GrayChannelValue>(
7        ((static_cast<unsigned int>(r)*4915 +
8          static_cast<unsigned int>(g)*9667 +
9          static_cast<unsigned int>(b)*1802) + 8192) >> 14);
10   }
11
12   inline GrayChannelValue
13   operator()(Image::PixelTriple pixel) const {
14     return operator()(pixel.red, pixel.green, pixel.blue);
15   }
16 };
```

Listing 8.1: RGB to Luminance Functor

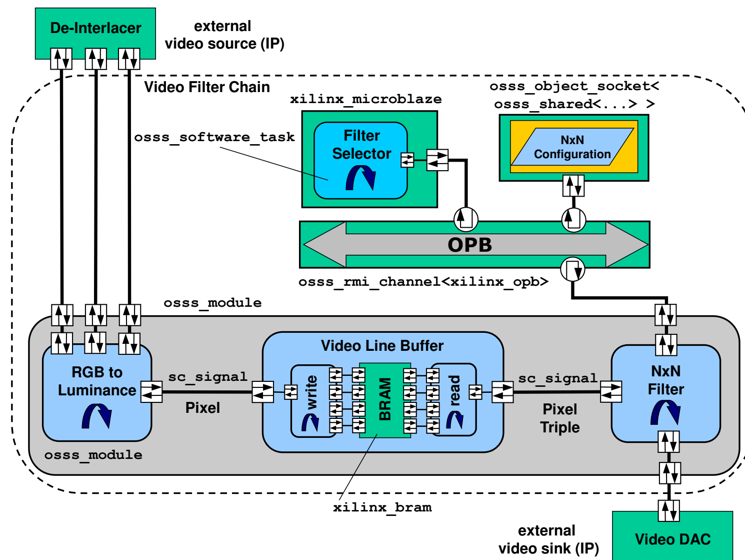
```
1  template<class Beh_t = osss::osss_behaviour>
2  class Color_Conversion_Beh : public Beh_t {
3  public:
4    osss_in <Image::PixelTriple> rgb_in;
5    osss_out<Image::pixel_t> gray_out;
6
7    void main() {
8      gray_out = m_cc(rgb_in);
9    }
```



(a) OSSS Behavior Layer Model



(b) OSSS Application Layer Model



(c) OSSS Virtual Target Architecture Layer Model

Figure 8.10: OSSS design flow applied to the adaptive video filter design [41]

```
10
11 protected:
12   RGB_To_Luminance<Image::pixel_t, Image::pixel_t>  m_cc;
13 };
```

Listing 8.2: Color Conversion Behavior

Listing 8.2 shows the RGB to luminance functor embedded inside the color conversion behavior. The separation of function/functor and behavior enables reuse of the function in different process contexts. I.e. the same functor will be used inside the OSSS Application Layer's hardware module (shown later).

Listing 8.3 shows a code snippet of the functional $N \times N$ filter implementation. The MAC (multiply accumulate) operation of the 2D convolution is implemented as an overloaded operator of the `Filter_Matrix` template class (see line 1-16). It is derived from a square matrix template class, which itself is implemented as a vector of vectors (see line 2). The `operator*` gets a coefficient matrix and performs the element-wise MAC operation.

```
1  template<class T, unsigned int dimension>
2  struct Filter_Matrix : public Vector<Vector<T, dimension>, dimension> {
3
4    template<typename Matrix_t>  Matrix_t
5    operator*(const Filter_Matrix<Matrix_t, dimension> &matrix) const {
6      Matrix_t result = 0;
7      for (unsigned int i=0; i<dimension; ++i) {
8        for (unsigned int j=0; j<dimension; ++j) {
9          Matrix_t coeff = matrix.get(i, j);
10         T pixel = this->get(i, j);
11         result += coeff * pixel;
12       }
13     }
14     return result;
15   }
16 };
17
18 template <typename pixel_t, typename coeff_t, unsigned int N>
19 class NxN_Filter {
20  public:
21   coeff_t operator()(const Vector<pixel_t, N >& data) {
22     pixel_matrix <<= 1;
23     pixel_matrix.setCol(0, data);
24     coeff_t result = pixel_matrix * coeff_matrix;
25     return (result >> Filter::COEFF_PRECISION) + shift;
26   }
27
28  protected:
29   coeff_t shift ;
30   Filter_Matrix<pixel_t, N> pixel_matrix;
31   Filter_Matrix<coeff_t, N> coeff_matrix;
32 };
```

Listing 8.3: $N \times N$ Filter Class

While the `Filter_Matrix` class defines a value type, the `NxN_Filter` class functor defines an entity type (see line 18-32), that is non-copyable and non-assignable. The `operator()` feeds a new line into the pixel matrix (left shift by one and column assignment), calls `operator*` from `Filter_Matrix`, performs a normalization and returns the filtered pixel.

```
1  template<class Beh_t = osss_behaviour>
2  class NxN_Filter_Beh : public Beh_t {
3   public:
4    osss_in<NxN_Configuration<Filter::coeff_t,
5      Filter_Matrix<Filter::coeff_t, Filter :: MATRIX_DIM> > > nxn_conf;
6    osss_in<Vector<Image::pixel_t, Filter::MATRIX_DIM > > pixel_row;
7    osss_out<Filter::coeff_t> filtered_pixel;
8
9    void main() {
10     filter .set_coeffs(nxn_conf);
11     filtered_pixel = filter (pixel_row);
12   }
```

```
13
14  protected:
15   NxN_Filter<Image::pixel_t, Filter::coeff_t, Filter :: MATRIX_DIM> filter;
16 };
```

Listing 8.4: $N \times N$ Filter Behavior

Listing 8.4 illustrates the implementation of the $N \times N$ filter behavior. It contains a port to the filter configuration Shared Variable (line 4-5), an input port for an incoming pixel vector (line 6) and an output port for the filtered pixel (line 7). The `main` routine reads the filter configurations from the Shared Variable and sets it as filter coefficients (line 10). The filter operation itself is performed by calling the `operator()` of `NxN_Filter` at line 11.

```
1  class Filter_Beh : public osss_behaviour {
2   public:
3    osss_in<NxN_Configuration<Filter::coeff_t,
4      Filter_Matrix<Filter::coeff_t, Filter :: MATRIX_DIM> > > nxn_conf;
5    osss_in<Image::PixelTriple> rgb_input;
6    osss_out<Filter::coeff_t> filtered_out;
7
8    Filter_Beh() {
9      color_conv.rgb_triple(rgb_input);
10     color_conv.gray_pixel(gray_pixel);
11     line_buffer.gray_pixel(gray_pixel);
12     line_buffer.pixel_row(pixel_row);
13     nxn_filter.pixel_row(pixel_row);
14     nxn_filter. filtered_pixel (filtered_out)
15     nxn_filter.nxn_conf(nxn_conf);
16   }
17
18   void main() {
19     osss_pipe( color_conv >> line_buffer >> nxn_filter );
20   }
21
22  protected:
23   osss_piped_variable<Image::pixel_t> gray_pixel;
24   osss_piped_variable<Vector<Image::pixel_t, Filter::MATRIX_DIM> > pixel_row;
25
26   Color_Conversion_Beh<osss_pipeline_stage> color_conv;
27   Cyclic_Buffer_Beh<osss_pipeline_stage> line_buffer;
28   NxN_Filter_Beh<osss_pipeline_stage> nxn_filter;
29 };
```

Listing 8.5: Video Filter Behavior

The video filter chain is implemented as a pipeline (see Listing 8.5) with the filter selector behavior running in parallel. The communication between the pipeline stages is performed by Piped Variables. Communication between the filter selector and the $N \times N$ filter is modeled using a Shared Variable. The video line buffer is not further described here. Its simple purpose it to safe N video lines (written pixel by pixel) and to deliver a column of N pixels to be processed by the $N \times N$ filter.

8.3.3.2 Application Layer Model

The next step is the refinement of the Behavior Layer Model to a synthesizable Application Layer Model (see Figure 8.10b). The Application Layer describes a synthesizable HW/SW partitioning. For this example we have decided to perform the video filter chain in hardware and the adaptive filter coefficient calculation and selection in software. Therefore, the filter selector behavior is converted into a Software Task.

All behavior blocks of the video filter pipeline have been converted into `osss_modules`. The piped variables have been converted to `sc_signals`. Listing 8.6 shows the OSSS Module implementation of the Video Filter Pipeline. The explicit pipeline of the OSSS Behavior Model has been replaced by a structural description of modules connected through signals. The double handshake interfaces of the RGB to luminance to the Video Source Stream and the $N \times N$ filter to the Video Stream Sink have been refined to signal level interfaces in order to get connected to a de-interlacer and a video DAC IP component.

```
1  class Filter_Module : public osss_module {
2   public:
3    osss_port<osss_shared_if<NxN_Configuration<Filter::coeff_t,
4      Filter_Matrix<Filter::coeff_t, Filter :: MATRIX_DIM> > > > nxn_conf;
5    sc_in<Image::pixel_t> r_input;
6    sc_in<Image::pixel_t> g_input;
7    sc_in<Image::pixel_t> b_input;
8    sc_out<Filter::coeff_t> filtered_out;
9
10   Filter_Module() {
11     color_conv.r(r_input);
12     color_conv.g(g_pixel);
13     color_conv.b(b_input);
14     color_conv.gray_pixel(gray_pixel);
15     line_buffer.gray_pixel(gray_pixel);
16     line_buffer.pixel_row(pixel_row);
17     nxn_filter.pixel_row(pixel_row);
18     nxn_filter. filtered_pixel (filtered_out)
19     nxn_filter.nxn_conf(nxn_conf);
20   }
21
22  protected:
23   sc_signal<Image::pixel_t> gray_pixel;
24   sc_signal<Vector<Image::pixel_t, Filter::MATRIX_DIM> > pixel_row;
25
26   Color_Conversion_Module color_conv;
27   Cyclic_Buffer_Module line_buffer;
28   NxN_Filter_Module nxn_filter;
29 };
```

Listing 8.6: Video Filter Module

Listing 8.7 shows the OSSS Module implementation of the $N \times N$ Video Filter. For the implementation, two hardware processes are required. The `config_proc` process is for polling and reading the filter configuration from the Shared Object. The `filter_proc` process performs the filter operation using the `NxN_Filter` functor. This filter class is reused from the Behavior Layer model.

```
1  class NxN_Filter_Module : public osss_module {
2   public:
3    osss_port<osss_shared_if<nxn_config_read_if> > nxn_conf;
4    sc_in<Vector<Image::pixel_t, Filter::MATRIX_DIM > > pixel_row;
5    sc_out<Filter::coeff_t> filtered_pixel;
6
7    OSSS_HAS_PROCESS(NxN_Filter_Module);
8
9    NxN_Filter_Module() {
10     SC_CTHREAD(config_proc, osss_module::clock_port.pos());
11     reset_signal_is(osss_module::reset_port, true);
12
13     SC_CTHREAD(filter_proc, osss_module::clock_port.pos());
14     reset_signal_is(osss_module::reset_port, true);
15   }
16
17  protected:
18   void config_proc() {
19     wait();
20     while(true) {
21       filter .set_coeffs(conf_in->get());
22       wait();
23     }
24   }
25
26   void filter_proc() {
27     wait();
28     while(true) {
29       filtered_pixel = filter (pixel_row);
30       wait();
31     }
32   }
```

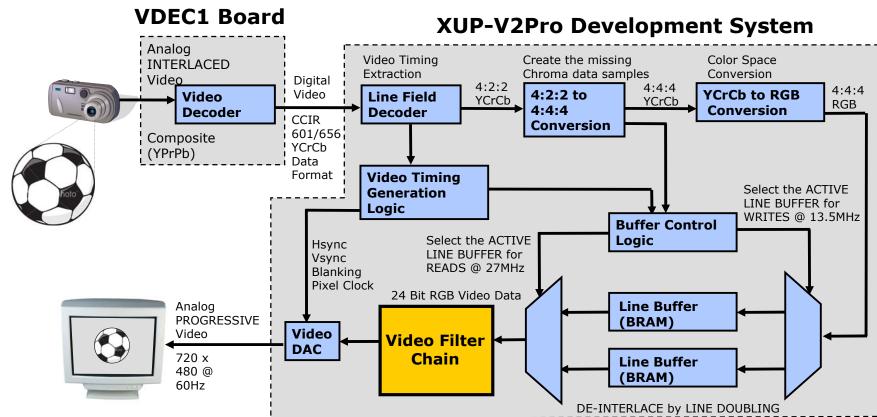


Figure 8.11: Xilinx Virtex-II FPGA implementation (Target Platform Layer) [41]

```
33
34      NxN_Filter<Image::pixel_t, Filter::coeff_t, Filter :: MATRIX_DIM> filter;
35    };
```

Listing 8.7: $N \times N$ Filter Module

8.3.3.3 Virtual Target Architecture Model

The final step is to map the Application Layer Model to a Virtual Target Architecture (see Figure 8.10c). The filter selector Software Task mapped onto a Xilinx MicroBlaze soft-core processor. The $N \times N$ filter configuration Shared Object is attached to the MicroBlaze's On-Chip Peripheral Bus (OPB). The communication from the software task to the Shared Object is implemented using the OSSS RMI Technology. To transport the `NxN_Configuration` via RMI it needs to be equipped with serialization support. Furthermore, the video line buffer is refined to make explicit use of a Xilinx Block RAM (BRAM) resource.

8.3.3.4 Target Platform Layer

The Virtual Target Architecture Layer Model is then automatically processed to a target technology dependent implementation using the synthesis flow from Figure 6.2. As shown in Figure 8.11 we have placed the synthesized video filter chain into a video processing environment on a Virtex-II Pro FPGA board (see Section E.1) equipped with a video decoder daughter board. Figure 8.12 shows the video filter FPGA implementation in action.

8.3.4 Conclusion

In this experiment, we have used the OSSS methodology for the design of a video filter and highlighted the following features: OSSS enables design and system synthesis of C++/SystemC models through a homogeneous system-level description language and simulation environment. It supports a stepwise and seamless refinement from a hardware/software independent executable model down to synthesizable hardware and software components and allows integration of pre-existing IP components. This ranges from RTL IP components with a cycle accurate pin interface, like RAMs, to system-level IP components, like SW processors and buses.

8.4 NightView Video Filter

8.4.1 Goals of this experiment

The NighView Video Filter is an extension of the Adaptive Video Filter design (see Section 8.3). It is an industrial design with high throughput and hard real-time requirements, realized in a

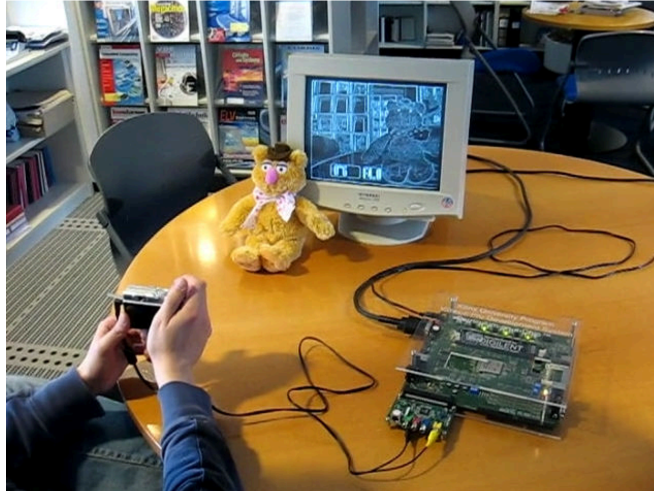


Figure 8.12: FPGA implementation of the video filter

mixed hardware/software design. The video processing is implemented as a hardware pipeline, while the configuration of this processing pipeline is implemented in software.

The goal of this experiment is to demonstrate

- the usage of Shared Objects for hardware/software communication,
- the Application to Virtual Target Architecture Layer mapping and refinement,

and to evaluate

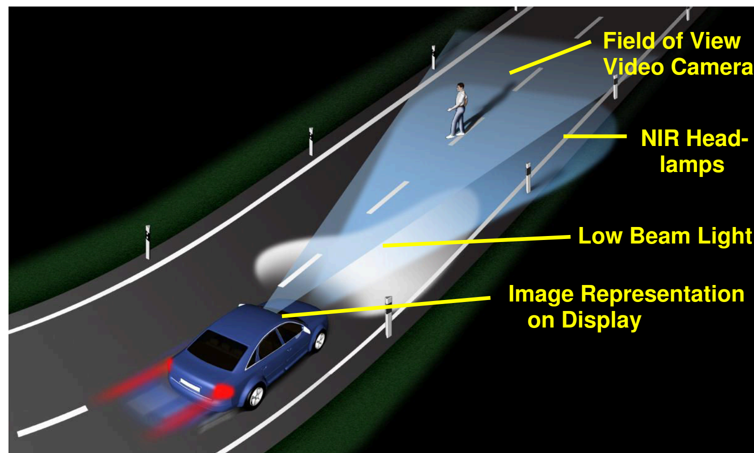
- the simulation performance of OSSS against VHDL,
- the OSSS model complexity against VHDL and
- the synthesis result in terms of area and critical path length (f_{max}) against manual optimized VHDL.

This experiment has been originally published in [62] and [29] (available in German only).

8.4.2 Introduction & Motivation

The presented design is an extract from the NightView video processing application, previously published in [29]. The purpose of the video processing is to bring the scene in front of a car onto a video display, located within the dashboard of a car. This supports the driver with additional video information and enhances the view compared to the information the driver can gain with the own eyes. Therefore, in certain situations like darkness, dawn, fog, etc., NightView allows a driver to detect critical situations earlier and to react faster than without the system. Figure 8.13 gives an overview of the NightView system.

NightView is not intended for a driver to exclusively watch the display instead of watching the street directly, but it helps the driver to early identify situations that are not clear yet and might become critical. The general idea of NightView is to use a camera to capture the area in front of a car. With an IR (Infra Red) camera that is very sensitive and specialized for dark scenes, a video stream is generated. The unmodified video stream is not well suited for a direct display. To create a satisfying video stream out of the raw data, major effort has to be done by several algorithms that are applied to modify the original pixel information to enhance the required scenes. As an output a video stream is generated that - compared to the direct drivers view - highlights dark scenes that are not in the driver's visible range. Also, in critical situations like other cars with high beam in contrary to the own car, the video system shall provide reliable information.



(a) Complete System Overview

(b) Comparison of the view with and without NightView

Figure 8.13: Overview of the NightView System [29]

8.4.3 The NightView Application

The original NightView system, from which the design example is derived, can be described in general as

1. an input stream (usually generated by a video camera),
2. several algorithms to enhance the video stream, and
3. the display of the final video output stream.

To be able to gain a consistent, standalone design example, the input and output components have to be included. However, the input does not necessarily need to be received from a camera. For simulation and test, a frame generator reads a frame sequence from a local disk providing the video input stream. Vice versa, the video output stream is simply dumped to the local disk. Figure 8.14 shows the general system structure of the NightView application.

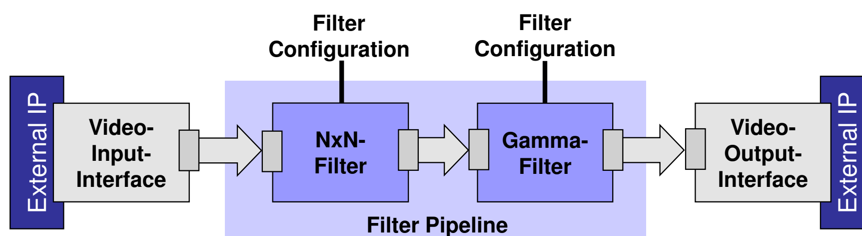


Figure 8.14: The general structure of the NightView application [62]

8.4.4 Target Platform

In order to provide a standalone application, the design target is an FPGA-based video application. The chosen target platform consists of the ML401 FPGA board [85], which is used for the evaluation purpose. Additionally, a VIODC video daughter board [85] providing the required interfaces for video input and output is connected to the ML401. The video daughter board has several digital and analogue video interfaces. To avoid several issues related to digital video processing like e.g. different pixel rates between video source and video sink, the digital interface (DVI) was chosen for video I/O. As a side effect, DVI sources like PC graphics cards in combination with TFT displays can be used and allows the usage of PC based multimedia applications such as DVD players and web cams. Figure 8.15 shows the ML401 board hook-up with the VIODC board.

Figure 8.15: The VIODC video daughter card connected to the ML401 board [62]

To gain access to the video interfaces, the Video Processing Evaluation Platform (VPEP) was developed to support rapid prototyping of various kinds of video applications. The VPEP is an IP core supporting the VIODC video daughter board described above and provides access to the incoming and outgoing video stream via an AEI bus [74].

8.4.5 Video processing algorithms

The core of the NightView application is the video processing pipeline. Two filters, a Sobel filter and a Gamma correction filter are successively applied to every single frame of the video stream. This architecture allows multiple configurations which results in different visual effects.

8.4.5.1 Sobel filter

The Sobel filter is an edge-detecting filter that increases the contrast edges. Given a Sobel configuration A containing the coefficients $a_{i,j}$, a two dimensional pixel array P with the sub-pixels $p_{i,j}$, and two shift values s_1 and s_2, the resulting pixel r_s is determined by:

$$r_s = \sqrt{r_1^2 + r_2^2}, \text{ with } r_1 = s_1 + \frac{1}{N}\sum_{i,j=0}^{N} p_{i,j}a_{i,j} \text{ and } r_2 = s_2 + \frac{1}{N}\sum_{i,j=0}^{N} p_{i,j}a_{N-j,i}$$

8.4.5.2 Gamma correction filter

The succeeding Gamma correction is needed to compensate the non-linear perception of the human eye. Additionally, the general contrast within a video frame is enhanced as well since the Sobel filter produces rather dark images. The gamma correction works on each pixel-value (usually provided as a look-up table) and modifis the pixel r_s in the following way:

$$r_\gamma = c \cdot r_s^\gamma$$

8.4.5.3 Filter configuration examples

The combination of two $N \times N$ filters and a Gamma correction filter allow multiple configurations by using different filter coefficients (see Table 8.3). Within this design example three different $N \times N$ filter coefficient combinations are used. The first matrix A_S is the edge detection matrix used for the Sobel filter:

$$A_S = \frac{1}{9} \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

The second matrix A_I is used for the pass-through of the unmodified central pixel:

$$A_I = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The third matrix is the null matrix A_0 with all coefficients set to zero. Different combinations of these three matrices lead to the three configurations used for the NightView design example. Table 8.3 gives and overview of these filter configurations.

	$N \times N$ filter 1 configuration	$N \times N$ filter 2 configuration
Plain	$A_1 = A_I$, $S_1 = 0$	$A_2 = A_0$, $S_2 = 0$
Emboss	$A_1 = A_S$, $S_1 = 2^7$	$A_2 = A_0$, $S_2 = 0$
Sobel	$A_1 = A_S$, $S_1 = 0$	$A_2 = A_S^T$, $S_2 = 0$

Table 8.3: Filter configurations used within the NightView design example

8.4.6 Design flow

To explore different design alternatives, the performed design flow was split into three sub paths (see Figure 8.16). The main path describes the flow of the actual NightView design example. Based on the specification (SystemC) of NightView (N1) several sub steps (N2a,b - N4) are performed resulting in the final design example prototype (N5). Path N2a-N3a consists of a high-level communication model. This high-level communication model has been refined to a more RTL-like communication scheme for resource efficient implementation (path N2b-N3b).

The second path leads to the VHDL reference design for which the same specification (N1) was used as for the first path. Some parts of the reference design, like the interfaces to video source and sink logic, where reused for the NightView OSSS prototype. The last path (V1 - V3) is the development of the VPEP-IP (Video Processing Evaluation Platform) [74] required by the final NightView design to access the DVI connectors of the VIODC.

The implemented interfaces were reused within the first and the second path. The first path (N2a + N3a) has not been considered for synthesis. All other paths lead to a synthesized design running on the FPGA target platform.

8.4.7 Modeling in OSSS

In the following sections only the path **N1** → **N2b** → **N3b** → **N4** → **N5** from Figure 8.16 will be presented. The other parts in Figure 8.16 (denoted by dashed boxes) were part of the design

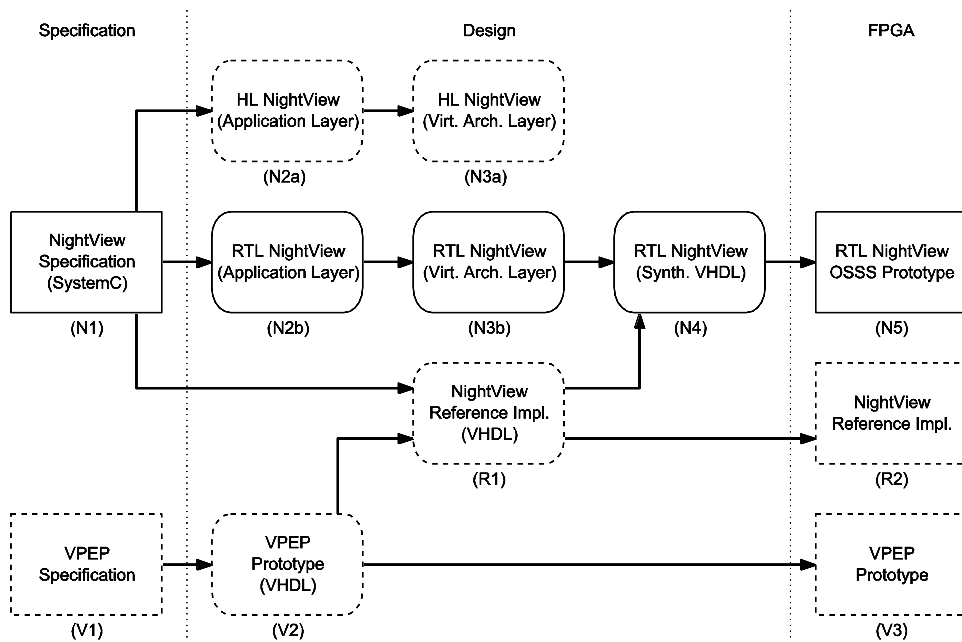


Figure 8.16: NightView modeling paths [62]

space exploration (N2a - N3b, R1 - R2) or were required to interface the video hardware (V1 - V3).

The following subsections will describe the Application Layer model (N2b) and the Virtual Target Architecture model (N3b). The next section will discuss the synthesis results (N4) and the reference implementation model in VHDL (R1). Furthermore, a comparison between the target platform model (N5) and (R2), both synthesized for the same FPGA, in terms of maximum clock frequency (critical path analysis) and area will be done.

8.4.7.1 Application Layer Model (N2b)

The NightView design has been modeled in a way that allows easy reconfiguration. Object-oriented features like encapsulation, templates, etc. have been used as often as possible. E.g. interfaces used for the description of Shared Objects are reused for various kinds of Shared Object implementations. Two general purpose interface descriptions have been introduced to model abstract read and abstract write accesses to a Shared Object. Both interfaces are parametrized by the read/write parameter/result type (see Listing 8.8) and are used for every Shared Object instance in the NightView design example.

```
1  template<class T>
2  class Reader_if : public virtual sc_interface {
3  public:
4    virtual T read() = 0;
5  };
6
7  template<class T>
8  class Writer_if : public virtual sc_interface {
9  public:
10   virtual void write(T &data) = 0;
11 };
```

Listing 8.8: The reader and writer interface used for all Shared Objects

Reuse also occurs for Shared Objects. A general Storage Shared Object is defined, which implements the reader and writer interface (see Listing 8.9). It stores a single value similar to a register but allows accessing it via method calls. The Storage class is parameterized over the storage type. This way it can be used for different kinds of storage classes. This Storage Shared

Object is used in the Top module where it stores the configuration data of the $N \times N$ and the Gamma correction filter.

```
1  template<class T>
2  class Storage : public virtual Reader_if<T>, public virtual Writer_if<T> {
3  public:
4    OSSS_GUARDED_METHOD_VOID(write, OSSS_PARAMS(1, T, item), true) {
5      currentItem   = item;
6       isInitialized  = true;
7    }
8
9    OSSS_GUARDED_METHOD(T, read, OSSS_PARAMS(0), isInitialized) {
10     return currentItem;
11   }
12
13 private:
14   T currentItem;
15   bool  isInitialized ;
16 };
```

Listing 8.9: Storage Shared Object

Parametrized user defined data types allow additional reuse. A vector class (see Listing 8.10) is used to model different kinds of array-like type of data. This class is used amongst others for the representation of filter coefficients, video data, and gamma look-up table.

```
1  template<class T, unsigned int dimension>
2  class Vector {
3  public:
4    Vector();
5    Vector(T  coefficients  []) ;
6    Vector(const Vector<T, dimension> &vector);
7    Vector<T, dimension> &operator=(const Vector<T, dimension> &vector);
8    T &operator[](sc_uint<MINIMUM_BIT_WIDTH(dimension) > columnIndex);
9    bool operator==(const Vector<T,dimension> &v);
10 private:
11   T data[dimension];
12 };
```

Listing 8.10: Vector class

Highlighting the configurability within OSSS, the bit width of the Vector class and other classes can be automatically kept as small as possible. By using template meta-programming techniques, the minimal bit width of counters can be determined during compile time (and respectively synthesis). See Listing 8.11 for the definition of the `MINIMUM_BIT_WIDTH` macro.

```
1  #define MINIMUM_BIT_WIDTH(X) MinimumBitWidth<X>::value
2
3  template <int N>
4  struct MinimumBitWidth {
5    enum {value = 1 + MinimumBitWidth<N/2>::value};
6  };
7
8  template <>
9  struct MinimumBitWidth<0> {
10   enum {value = 0};
11 };
```

Listing 8.11: Template meta-programming to determine the minimal required bit width

Top-Level Module The Top module encapsulates the core modules of the NightView system (see Figure 8.17). The NightView system is required to enable real-time video processing. This leads to the following timing constraint: Since a minimal resolution of 640×480 pixel should be processed at a frame refresh rate of 60 Hz the total pixel throughput must be about 25 MHz. A possible way to process such large amounts of data is to introduce a processing pipeline such that the different processing steps can be done in parallel.

Three modules that buffer the video stream in two dual-ported RAM modules and handle the translation of the video stream to and from the filter pipeline and the AEI bus act as

the interface to the actual video hardware. Incoming and respectively outgoing pixels are not handled directly by the video-processing pipeline. They are stored within the two RAM modules, which allow the whole NightView application to operate as an AEI slave (which is required by the VPEP IP). Data access width to the RAM is always 32 bit to match the AEI payload size. Therefore, four pixel chunks are read and respectively written simultaneously.

The Source and the Sink Transactor provide the access to the two RAM modules from the pipeline. The Source Transactor translates multiple pixel chunks to pixel columns containing N pixels, which can be transferred to the Filter Pipeline. The Sink Transactor collects incoming pixels until 4 of them can be combined into a single chunk which can be afterwards written to the RAM.

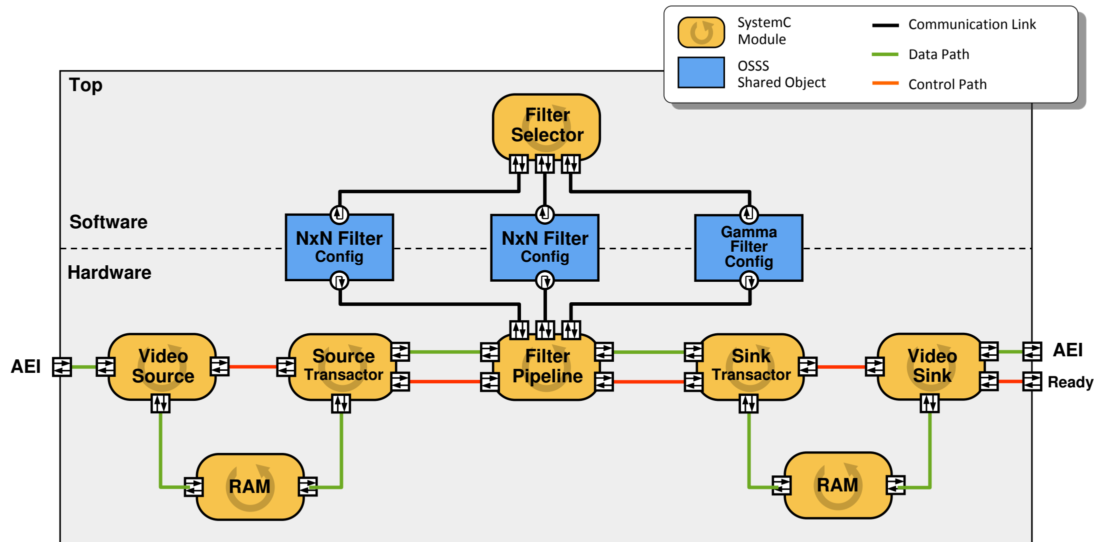


Figure 8.17: Top-Level Module of the RTL Application Layer Model (N2b) [62]

Testbench Inside the testbench (see Figure 8.18) of the NightView design, the input video stream is read from the local file system. The Video Producer sequentially reads single image files (PGM graphics file) and transfers video lines via the AEI bus to the Video Source in the Top module. The result of the enhancements is requested by the Video Consumer that stores the received video frames back in the local file system. The Video Consumer is connected to the Top module via a second AEI bus.

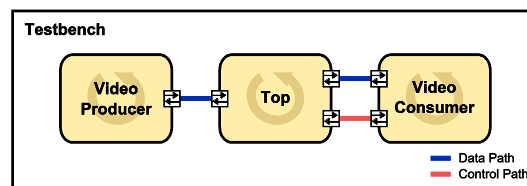


Figure 8.18: Testbench of the RTL Application Layer model (N2b) [62]

Filter Pipeline The core of the NightView design is the Filter Pipeline module (see Figure 8.19). It mainly consists of two $N \times N$ filters and one Gamma filter. The Filter Pipeline can be configured by different coefficients to meet different needs due to changing environment conditions (like dusk, night, etc.). The decision of which filter is used is done within software. In the NightView design a user can select different configurations by using the keypad on the ML401. The communication between the software and the hardware part is done via Shared Objects (`NxN Filter Config` and `Gamma Filter Config`).

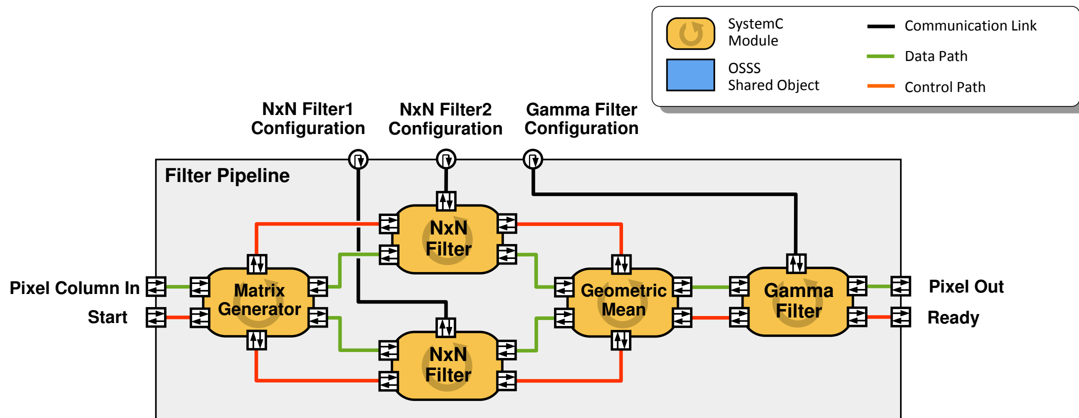


Figure 8.19: Filter Pipeline of the RTL Application Layer model (N2b) [62]

To perform the Sobel operation the filter needs an $N \times N$ filter matrix. The filter pipeline receives an N pixel column as input and outputs a single pixel. The configurations for the $N \times N$ filters and the Gamma filter are accessed by the filter pipeline via OSSS Shared Objects. Since the Shared Objects themselves are part of the top module, a simple port-to-port binding connects the two filters with the enclosing Top module.

The N pixel columns are sequentially received and stored within a shift register (Matrix Generator). After the first N columns in a video line are received, the first $N \times N$ matrix is complete. The matrix can be now transmitted to the two $N \times N$ filters that perform the core Sobel operation. Afterwards, the two resulting pixels of the $N \times N$ filters are combined together via the Geometric Mean module. The combined pixel is finally modified by the Gamma correction filter. After the first N columns of a video line are received, the filter pipeline processes one pixel per clock cycle.

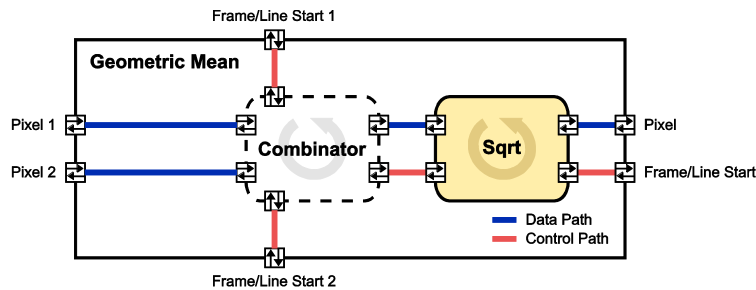


Figure 8.20: Geometric Mean Module [62]

The Geometric Mean module combines the two resulting pixels from the $N \times N$ filters (Combinator) into a single pixel by determining the geometric mean (see Figure 8.20). The main function of the geometric mean is the calculation of a square root. To obtain the square root of s, the Newton iteration (see Equation 8.4.7.1) is applied with a fixed iteration count of 10 (10 iterations deliver sufficiently precise results in this design example).

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ with } f(x_n) = x_n^2 - s$$

In order to enable a throughput of one pixel per clock cycle the required calculation of the square root is organized as a sub-pipeline (see Figure 8.21). This sub-pipeline consists of the 10 stages whereas each step calculates one step of the Newton iteration. Each step receives the actual parameter of the square root and the actual intermediate value of the iteration and returns the new intermediate value.

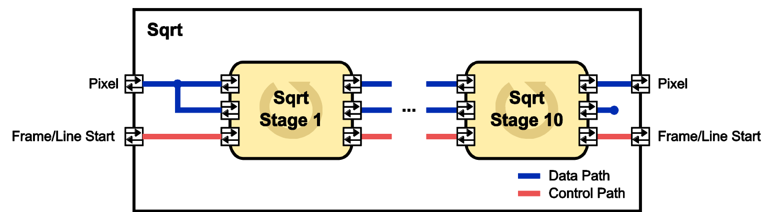


Figure 8.21: Sqrt Module [62]

The first and the last stage of the sub-pipeline need some special care: The first stage receives the same value for the intermediate value because no intermediate value is obtained at this point. The output parameter of the last stage is not used and is simply connected to a dummy signal. Finally yet importantly, the control signals indicating frame/line start are forwarded as well. Due to the 10 iteration steps, the latency of the Sqrt module is 10 clock cycles.

8.4.7.2 Virtual Target Architecture Layer Model (N3b)

The OSSS methodology allows successively transforming the existing design from the Application Layer to the Virtual Target Architecture Layer. Using the same testbench the design can be validated against the golden specification model after every transformation step.

In the following a brief description of the NightView architecture depicting the OSSS related differences to the Application Layer is given.

The top-level module (see Figure 8.22) integrates all hardware modules and the software task within a single module. Communication between hard- and software is realized through Shared Objects (similar to the top-level module on the Application layer). The connection to the video streams is obtained by providing ports to the AEI bus of the VPEP (and its VHDL implementation).

Two dual-ported RAM modules buffer the video stream and handle the translation of the video stream to and from the filter pipeline and the AEI bus. To connect the NightView system with the actual video input and output stream, the Video Source and Sink modules handle the access to the AEI bus.

Incoming and respectively outgoing pixels are not handled directly by the video processing pipeline. They are stored within the two RAM modules, which allow the whole NightView application to operate as an AEI slave (which is required by the VPEP IP). In order to keep the bandwidth low, the data access to the RAM is always 32 bit wide. Therefore, four pixel chunks are read and respectively written simultaneously.

Two modules, the Source and the Sink Transactor, provide the access to the two RAM modules from the pipeline. The Source Transactor translates multiple pixel chunks to pixel columns containing N pixels which can be transferred to the Filter Pipeline. The Sink Transactor collects incoming pixels until 4 of them can be combined into a single chunk which can be afterwards written to the RAM.

```
1  class Top : public xilinx_system {
2    sc_in<bool> clock_port;
3    sc_in<bool> reset_port;
4
5    // aei ports to video source
6    sc_in<aei_sel_t>     aeiSelectSource;
7    sc_out<aei_ready_t> aeiReadySource;
8    sc_in<aei_w1r0_t>    aeiWriteEnableSource;
9    sc_in<aei_byteen_t> aeiByteEnableSource;
10   sc_in<aei_addr_t>    aeiAddressSource;
11   sc_in<aei_data_t>    aeiDataSource;
12
13   // aei ports to video sink
14   // ...
15
16   // Configuration Shared Objects
17   osss_object_socket<osss_shared<Storage<nxnconfig_t> > > *nxnConfig;
```

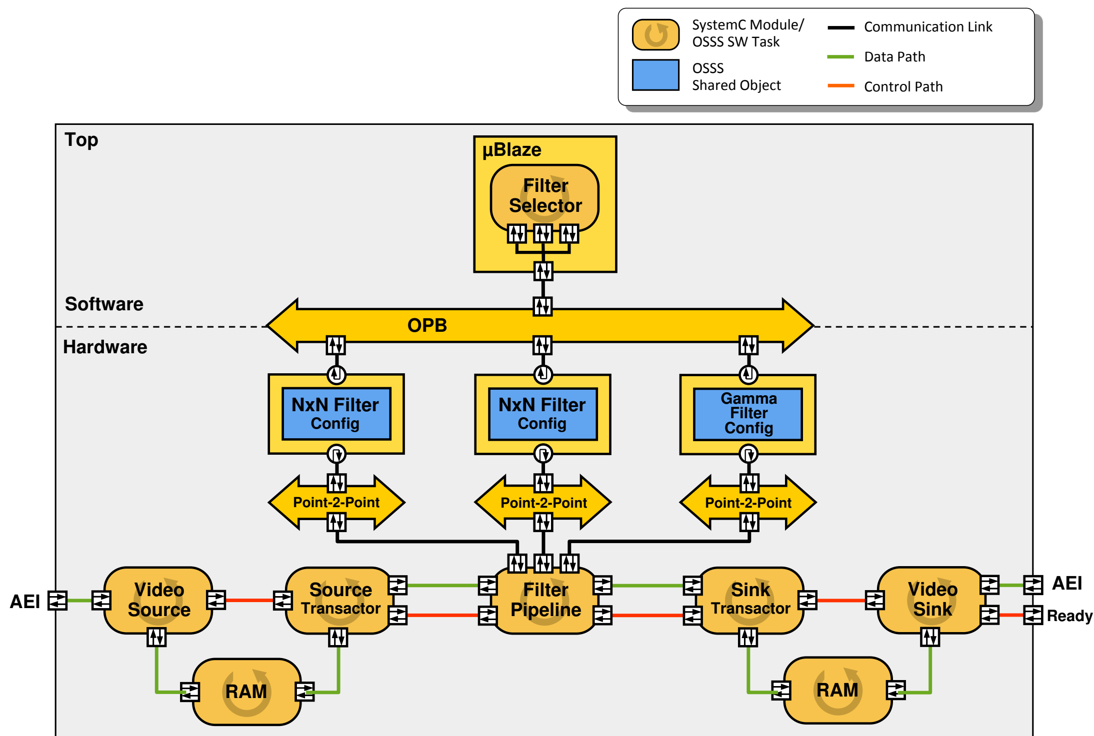


Figure 8.22: The Top module of the NightView system on Virtual Target Architecture Layer (N3b) [62]

```
18  osss_object_socket<osss_shared<Storage<lookup_t> > >   *gammaConfig;
19
20  // ...
21
22  SC_HAS_PROCESS(Top);
23
24  Top(sc_core::sc_module_name module_name) : xilinx_system(module_name) {
25    CREATE_OSSS_CHANNEL(mbRMIChan,     hwsw_channel_t, "mbRMIChan");
26    CREATE_OSSS_CHANNEL(nxnConf1Chan,  hwhw_channel_t, "nxnConf1Chan");
27    CREATE_OSSS_CHANNEL(nxnConf2Chan,  hwhw_channel_t, "nxnConfig2Chan");
28    CREATE_OSSS_CHANNEL(gammaConfChan, hwhw_channel_t, "gammaConfChan");
29
30     filterPipeline ->nxnConfiguration1Reader(*nxnConf1Chan,   nxnConfig1);
31     filterPipeline ->nxnConfiguration2Reader(*nxnConf2Chan,   nxnConfig2);
32     filterPipeline ->gammaConfigurationReader(*gammaConfChan, gammaConfig);
33
34    nxnConfig1.bind(*mbRMIChan);
35    nxnConfig1.bind(*nxnConf1Chan);
36    nxnConfig2.bind(*mbRMIChan);
37    nxnConfig2.bind(*nxnConf2Chan);
38
39    gammaConfig.bind(*mbRMIChan);
40    gammaConfig.bind(*gammaConfChan);
41
42    microProcessor = new xilinx_microblaze("microProcessor");
43    microProcessor->rmi_client_port(*mbRMIChan);
44    microProcessor->add_sw_task(filterSelector);
45    microProcessor->clock_port(clock);
46    microProcessor->reset_port(reset);
47
48    microblazeRMIChannel->clock_port(clock);
49    microblazeRMIChannel->reset_port(reset);
50
51    nxnConfig1Channel->clock_port(clock);
52    nxnConfig1Channel->reset_port(reset);
53    // ...
```

```
54   }
55
56   // ...
57 };
```

Listing 8.12: Top-level module of the Virtual Target Architecture Layer

The former definition of the configuration Shared Objects are adapted to reflect the RMI style communication on the Virtual Target Architecture Layer. Therefore, the definitions of the Shared Objects are changed to `osss_object_sockets`. Additionally, all Shared Object communications have to be mapped to a specific bus. The communication to the Filter Selector software task is mapped to the OPB whereas all other communications are mapped to point-to-point channels (see Figure 8.22 and Listing 8.12).

In order to allow RMI, all data transmitted have to be serialized such that the contained data can be transmitted over the OPB respectively point-to-point channels. E.g. the Vector class is modified as shown in Listing 8.13. The class itself is now derived from `osss_serialisable_object` and has been tagged as serializable by the `OSSS_IS_SERIALISABLE` macro. Additionally, all constructors have been wrapped by the `OSSS_SERIALISABLE_CTOR` macro. The core of the serialization are the methods `serialise()` and `deserialise()` which perform the actual serialization and deserialization of the object.

```
1  template<class T, unsigned int dimension>
2  class Vector : public osss_serialisable_object {
3  public:
4    OSSS_IS_SERIALISABLE(Vector);
5
6    OSSS_SERIALISABLE_CTOR(Vector, ()) {}
7
8    OSSS_SERIALISABLE_CTOR(Vector, (T coefficients[])) {
9      for (sc_uint<MINIMUM_BIT_WIDTH(dimension)> i=0; i<dimension; i++)
10       data[i.to_int()] = coefficients [i.to_int()];
11   }
12
13   OSSS_SERIALISABLE_CTOR(Vector, (const Vector<T, dimension> &vector)) {
14     for (sc_uint<MINIMUM_BIT_WIDTH(dimension)> i=0; i<dimension; i++)
15       data[i.to_int()] = vector.data[i.to_int()];
16   }
17
18   // ...
19
20   virtual void serialise () {
21     for (int i = 0; i < dimension; i++)
22       store_element(data[i]);
23   }
24
25   virtual void deserialise() {
26     for (int i = 0; i < dimension; i++)
27   restore_element(data[i]);
28   }
29
30   // ...
31 };
```

Listing 8.13: Serialization of the Vector class

Other user defined data types, which are also transmitted via RMI, as the NxNConfiguration class (that stores the NxN coefficients and the shift value) needs to be equipped with serialization support in a similar way.

8.4.8 Evaluation

This section describes the evaluation of the NightView design example regarding simulation performance, code quality, and efficiency of the RTL synthesis. In this evaluation, only the Filter Pipeline module will be considered for comparison with the NightView reference implementation (R1).

8.4.8.1 Simulation performance

The different designs presented in Figure 8.16 have been tested with different still picture and video sequences. For an efficient exploration of many different models, an adequate simulation speed is necessary.

Table 8.4 shows the simulation time of four selected NightVision implementation versions. The measured time is the duration of the simulation for filtering an image of the dimension 720×480. Both OSSS models N2b and N3b have been compiled with the Accellera SystemC 2.2.0 kernel. The VHDL models R1 and N4 have been simulated using Mentor Graphics ModelSim 6.1e. All simulations have been performed on a T7600 (Intel Core 2 Duo) running at 2.33 MHz with 1 GB RAM.

The *Fossy* generated code has been tested with the same testbench as for model N3b to obtain comparable simulation results. The resulting simulated execution time of the filter pipeline is 12.3 ms per frame running at a clock frequency of 25 MHz. This results in a throughput of 83 frames per second and leaves enough headroom since the specification only requires 60 frames per second.

Comparing the simulation times of the OSSS Application Layer and Virtual Target Architecture Layer models with the VHDL reference or *Fossy* generated VHDL model shows that algorithmic exploration and refinement in OSSS is more efficient than in VHDL. Even the VTA model at RTL is more than 3 times faster without losing any accuracy.

Comparing simulation times between the hand-written VHDL model (R1) and the *Fossy*-generated model (N4) an overhead of about 20 % in model N4 can be observed for this design example. A major source of simulation overhead in the *Fossy* generated VHDL model (N4) is the SystemC to VHDL integer type mapping as described in Section 7.7. The introduced conversion functions to retain the SystemC integer type semantics in VHDL can be elaborated at VHDL model compile time, but induce additional overhead compared to a native VHDL integer type usage as done in model R1.

Version (Simulator)	Simulation time [s]
Application Layer N2b (SystemC 2.2.0)	76.8
Virtual Target Architecture Layer N3b (SystemC 2.2.0)	123.7
VHDL Reference R1 (ModelSim 6.1e)	413.4
VHDL *Fossy* generated N4 (ModelSim 6.1e)	495.1

Table 8.4: NightView design simulation times [62, 29]

8.4.8.2 Model complexity

In order to evaluate the model complexity of the generated VHDL code (N4) a comparison with the hand-written VHDL version (R1) of the NightView design has been performed. Since the hand-written version does contain neither the MicroBlaze subsystem nor any hardware/software communication only the video filter pipelines module will be considered again.

However, an objective comparison only based on lines of code metric is not adequate. Different coding styles, length of lines, etc. largely influence the number of code lines. Especially for generated code a lines of code metric is not appropriate. The number of code lines shall only be used as a first indicator here.

Version	Lines of Code without comments (Language)
Application Layer N2b	2859 (OSSS/SystemC)
Virtual Target Architecture Layer N3b	3250 (OSSS/SystemC)
VHDL Reference R1	3442 (VHDL)
VHDL FOSSY generated N4	2667 (VHDL)

Table 8.5: NightView design comparisons Lines of Code [62, 29]

The refinement of the Application Layer design (N2b) to the Virtual Target Architecture Layer design (N3b) requires 391 lines of code. This step could be automated by a tool, as is consists in adding serialization support to value classes, replacing ports (RMI stub insertion), configuration of the target platform, and mapping application layer elements to the target platform resources. The VHDL model N4 that is generated from model N3b has less lines of code. The reason of this reduction in lines of code is that the Application Layer model as well as the Virtual Target Architecture Layer model have been written for reuse. Functions, classes, and processes have been defined for different possible configurations and instantiations. The generated VHDL code only contains a dedicated configuration and instantiation of the model. Code that is not used in this configuration does not result in generated code in model N4.

Comparing the hand written reference implementation R1 with the *Fossy* generated model N4 we can observe that the number of code lines in the reference model is significantly higher than in the generated model. The main reason is that the hand written VHDL model uses different functions and procedures that have been designed for generic reuse in different VHDL signal processing designs. Moreover, certain coding guidelines (for better code maintainability and re-use) applied to the reference design result in an increase of lines of code.

8.4.8.3 Chip area

Table 8.6 shows that the synthesis results generated by FOSSY are about in the same range as the hand-written version. Area usage and estimated frequencies are close to each other's. A minor discrepancy might also come from a slightly different implementation of the hand-written version.

Both designs have been synthesized using the Xilinx Synthesis Tool (XST) 8.2 targeting a Virtex-4 FPGA. In this design example, the *Fossy* generated VHDL design needs 16% more resources (measured as total equivalent gate count) and has a reduced maximum clock frequency of 3%.

The main source of higher resource consumption of the *Fossy* generated VHDL design is the SystemC to VHDL integer type mapping. The design makes excessive use of integer arithmetic inside the $N \times M$ filters, and the unrolled Newton Iteration for the square root approximation inside the geometric mean module. The implicit size extensions of SystemC integers, as described in Section 7.7 comes at the cost of 16% more area (this can be split-up into 36% more flip-flops for integer type storage elements and 10% more combinatorial overhead). Since the overall FPGA area utilization is quite low in this example, the critical path and thus the maximum estimated frequency $f_{\max}$ is only reduced by 3%.

	VHDL Reference (R1)	VHDL *Fossy* (N4)	Delta
Number of slice flip-flops	441	598	36%
Number of 4 input LUTs	2,098	2,312	10%
Total equivalent gate count	21,131	24,531	16%
Estimated frequency ($f_{\max}$) [MHz]	78.1	75.7	-3%

Table 8.6: NightView design synthesis results [62, 29]

8.4.9 Conclusion

In the NightView example, the OSSS methodology has been applied to an industrial video processing application. The simulation of the Application Layer model runs more than 5 times faster than the VHDL simulation of the handcrafted VHDL design. Even the Virtual Target Architecture model still runs more than 3 times faster than the VHDL model. From the modeling point of view, the Application Layer model requires 20% less lines of code for the same functionality with increased reuse possibilities. OSSS Application Layer code can also be reused for software implementations for instance. The Virtual Target Architecture Layer model's complexity is very similar to the VHDL model complexity. The VHDL model has 5% more lines of code. The synthesis of the Virtual Target Architecture Layer model leads to a 16%

overhead in chip area and a reduction of the maximum clock frequency of only 3% compared to the manual optimized VHDL implementation.

8.5 MP3 Decoder

8.5.1 Goals of this experiment

The goal of this experiment is an assessment of the C++ and RMI protocol overhead for hardware/software communication. For this purpose a comparison with a state-of-the-art optimized C register-based memory mapped I/O communication is performed. This experiment has been conducted in an MP3 decoder design. The very compute intense Discrete Cosine Transform (DCT) of the subband synthesis step in the decoding chain has been implemented in hardware: As regular VHDL module with register interface and as OSSS Shared Object. A comparison of the different DCT execution times and communication overhead is performed. This experiment has been performed originally in [66].

8.5.2 Introduction to MP3 decoding

Figure 8.23 provides an overview of the block structure of an MP3 decoder. The used blocks are briefly explained in the following subsections.

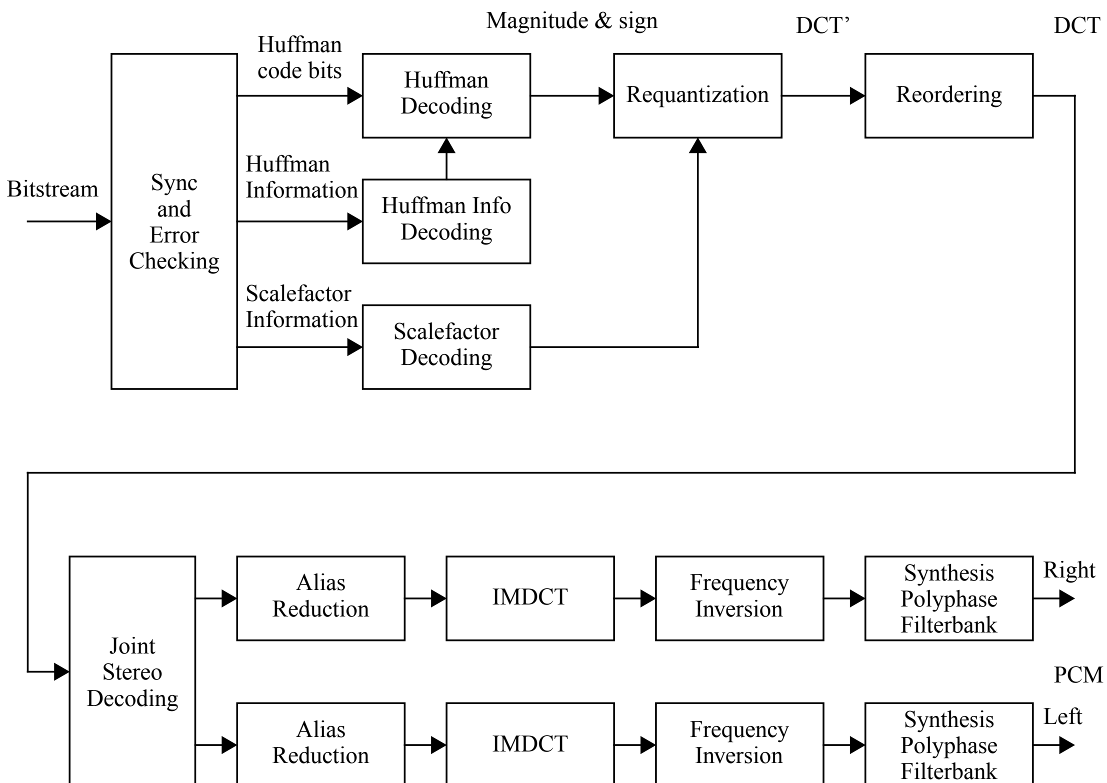


Figure 8.23: MP3 decoder structure [178]

Sync and Error Checking This block receives the encoded MP3 bitstream. Every frame within the stream is identified by searching for the synchronization word. It is not possible for the following processing blocks to extract the correct information without correct frame detection. Error checking recomputes the CRC checksum of each frame and compares it with the stored checksum.

Huffman Info Decoding & Huffman Decoding Huffman coding is a variable length coding method. For this reason, the start of the code word needs to be identified. This information is provided by the Huffman info decoding block. The purpose of this block is to provide all necessary parameters to the Huffman decoding block to perform a correct decoding. Moreover, the Huffman info decoder block must insure that 576 frequency lines are generated regardless of how many frequency lines are described in the Huffman code bits. If less than 576 frequency lines appear, the Huffman info decoding block adds zero data (padding).

Scalefactor Decoding This block decodes the coded scalefactors, i.e. the first part of the main data. The scalefactor is obtained from the side information. The decoded scalefactors are used for requantization.

Requantization The decoded, scaled and quantized frequency lines output from the Huffman decoder block are requantized using the scalefactors together global gain and preflag field information.

Reordering In order to increase the efficiency of the Huffman coding, the frequency lines for the short window cases were reordered into subbands first, then frequency and at last by windows by the encoder. Samples close in frequency are more likely to have similar values and thus can be much better Huffman encoded. The reordering block will search for short windows in each of the 36 subbands. If short windows are found they are reordered.

Joint Stereo Decoding The purpose of the Joint Stereo Decoding block is to perform the necessary processing to convert the encoded stereo signal into separate left/right signals. The method used for encoding the stereo signal can be read from the mode and mode extension in the header of each frame.

Alias Reduction The alias reduction is required to negate the aliasing effects of the polyphase filterbank in the encoder. The alias reconstruction calculation consists of eight butterfly calculations for each subband.

Inverse Modified Discrete Cosine Transform (IMDCT) The frequency lines from the Alias Reduction block are mapped to 32 Polyphase filter subbands. The IMDC will output 18 time domain samples for each of the 32 subbands.

Frequency Inversion In order to compensate for frequency inversions in the synthesis polyphase filter bank, every odd time sample of every odd subband is multiplied with -1.

Synthesis Polyphase Filterbank The Synthesis Polyphase Filterbank transforms the 32 subbands of 18 time domain samples in each granule to 18 blocks of 32 PCM samples, which is the final decoding result. This filterbank uses a 32-point Discrete Cosine transform (DCT) to perform subband synthesis efficiently. The implementation efficiency of this filterbank is very important. In dependance on the sampling rate of the audio signal, the filterbank is executed 3000 times per second (e.g. for $f_{\text{sample}} = 48$ kHz).

8.5.3 Modeling in OSSS

The OSSS model is based on the MAD (MPEG audio decoder) decoder [227]. It supports MPEG-1 and the MPEG-2 extension to lower sampling frequencies, as well as the de facto MPEG 2.5 format. All three audio layers – Layer I, Layer II, and Layer III (i.e. MP3) – are fully implemented. MAD has the following special features:

- supports 24-bit PCM output
- 100% fixed-point (integer) computation
- implementation based on the ISO/IEC standards
- available under the terms of the GNU General Public License (GPL)

8.5.3.1 Profiling

We start with a profiling of the MAD decoder on our Xilinx MicroBlaze Target processor.

```
---------Simulation Statistics---------
                 Cycles : 22382061
           Instructions : 7033278
                    CPI : 3.182
                  Loads : 2070903
                 Stores : 748313
        Multiplications : 749741
              Divisions : 470
          Barrel shifts : 482772
    Total # of branches : 355537
    # of taken branches : 269015 (75.664%)
                  Calls : 17771
                Returns : 17771
        ICache Accesses : 7363343
            ICache Hits : 7356486 (99.907%)
        DCache Accesses : 2070332
            DCache Hits : 1524544 (73.638%)
----------Pipeline Stall Info-----------
      Total # of stalls : 17344977
          IFetch stalls : 340728 (1.964%)
   Memory Access stalls : 14993995 (86.446%)
      Multiplier stalls : 1499482 (8.645%)
  Barrel Shifter stalls : 495732 (2.858%)
         Divider stalls : 15040 (0.087%)
--------------Branch Info---------------
Branches w/  delay slot : 207268 (1.195%)
Branches w/o delay slot : 123494 (0.712%)
```

Listing 8.14: Simulation statistics of MAD decoder execution on a Xilinx MicroBlaze ISS [66]

Listing 8.14 shows the result of the statistics of the MAD decoder running on a Xilinx MicroBlaze Instruction Set Simulator (ISS). The data has been obtained from decoding 29 frames with a bit rate of 32 kb/s and a sampling frequency of 44,1 kHz. As show in the statistics, the decoding takes 22.382.061 clock cycles. When running the processor with a frequency of 100 MHz the corresponds to approximately 223,82 ms. This is clearly below the upper timing bound of 757,55 ms for seamless real-time decoding.

In addition to the above run-time statistics, the number of executed instructions, the number of multiplications and divisions, and the number of cache hits are reported. From these statistics, the configuration of the MicroBlaze processor can be adapted. The reported high number of multiplications and barrel shifts demand for a dedicated hardware multiplier and special barrel shifter hardware component to speed-up the decoding.

```
Each sample counts as 1e-08 seconds.
 %    cumulative   self              self     total
time   seconds   seconds   calls   us/call   us/call   name
54.15     0.11      0.11    2088     54.02     54.02   dct32
18.50     0.15      0.04      29   1329.01   5218.27   synth_full
 7.90     0.17      0.02     604     27.24     33.43   III_imdct_l
 7.34     0.18      0.02      29    527.46   1858.80   mad_layer_III
 3.82     0.19      0.01     108     73.71     73.71   III_aliasreduce
 1.79     0.19      0.00    1208      3.09      3.09   fastsdct
 1.23     0.20      0.00    1856      1.38      1.38   III_freqinver
 1.17     0.20      0.00    4743      0.51      0.51   mad_bit_read
 1.05     0.20      0.00     666      3.27      3.27   III_overlap
 0.72     0.20      0.00      62     24.31     24.31   III_imdct_s
[...]
```

Listing 8.15: Profiling results of MAD decoder execution on a Xilinx MicroBlaze ISS (excerpt) [66]

Listing 8.15 shows the profiling results for the same bit stream as used in Listing 8.14. The profiling reports detailed information for each function of the executed program: the number calls and their duration. From the profiling run it becomes obvious, that the overall runtime of

the MP3 decoder is dominated by the dct32 function (54,15%) used in the subband synthesis. In the Application Layer Model the DCT is realized as Shared Object and implemented in hardware to speed-up the overall decoding process.

8.5.3.2 Application Layer Model

```
1  class Top : public osss_system {
2    sc_in<bool> Clk;
3    sc_in<bool> Reset;
4
5    player *player1;
6    osss_shared< dct > *dct_inst;
7
8    Top() {
9      dct_inst = new osss_shared<dct>("dct1");
10     dct_inst->clock_port(Clk);
11     dct_inst->reset_port(Reset);
12     player1 = new player("player1");
13     player1->clock_port(Clk);
14     player1->reset_port(Reset);
15     player1->dct_port(*dct_inst);
16   }
17 };
```

Listing 8.16: Top-Level module of the Application Layer model

The Application Layer model consists of a single Software Task `player` and a Shared Object implementing the DCT. Listing 8.16 shows the top-level module of the Application Layer model. In this design, the Shared Object is used by a single client only and provides a method-based interface to the DCT functionality.

```
1  class player : public osss_software_task {
2  public :
3   osss_port_to_shared< dct_if > dct_port;
4
5   OSSS_SW_CTOR(player) { }
6    ...
7
8   void dct32(mad_fixed_t const in[32], unsigned int slot,
9              mad_fixed_t lo[16][8], mad_fixed_t hi[16][8]) {
10    int i = 0;
11    dct_data input, output;
12    // preparation of the dct data object
13    for(i = 0; i < 32; i++) {
14      input.setData(i, in[i]);
15    }
16
17    // call to the DCT Shared Object
18    output = dct_port->do_dct(input);
19
20    // unpacking of the return data object
21    for(i = 0; i < 16; i++) {
22      hi[15-i][slot] = output.getData(i);
23      lo[i][slot] = output.getData(i+16);
24    }
25  }
26   ...
27 };
```

Listing 8.17: MP3 player Software Task with DCT Shared Object call

Listing 8.17 shows an excerpt of the `player` Software Task where the call of the DCT Shared Object happens. The `dct32` function (line 8) is called in the subband synthesis of the MAD decoder software. Profiling has shown, that this function consumes over 50% of the decoding time of a frame. By moving the DCT from software to a hardware implementation inside a Shared Object a certain speed-up is expected, assuming the hardware DCT computes faster than in software, including the hardware/software communication overhead. Before calling the `do_dct` function of the Shared Object (line 18) the DCT data gets stored into an object. After

completion of the DCT, the return data object is unpacked in the same way. The presented implementation is simple and does not performed pipelined execution (e.g. separate DCT send and return functions) because the goal of this experiment is measuring the RMI communication overhead, not presenting the most efficient implementation.

```
1  class dct_data {
2  public:
3    dct_data() {}
4    void setData(int pos, sc_uint<32> value) { m_data[pos] = value; }
5    sc_uint<32> getData(int pos) { return m_data[pos]; }
6  private:
7    sc_uint<32> m_data[32];
8  };
9
10 class dct_if : public sc_interface {
11 public:
12   virtual dct_data do_dct(dct_data in) = 0;
13 };
14
15 class dct : public dct_if {
16 public:
17   CLASS(dct, NullClass);
18   CONSTRUCTOR(public, dct, ());
19   OSSS_GUARDED_METHOD(dct_data, do_dct, OSSS_PARAMS(1, dct_data, in), true);
20 };
```

Listing 8.18: DCT Shared Object, with interface declaration and data object definition

Listing 8.18 shows the DCT data object `dct_data`, the DCT Shared Object interface `dct_if` and the DCT Shared Object implementation `dct`. Since the `do_dct` service is executed atomically, no guard condition is specified (i.e. the guard condition is always true).

8.5.3.3 Virtual Target Architecture Layer Model

```
1  class Top : public xilinx_systsm {
2  public:
3    sc_in<bool> Clk, Reset;
4
5    player *player1;
6    osss_rmi_channel<xilinx_opb_channel<false, false> > *channel;
7    osss_object_socket <osss_shared< dct > > *dct_inst;
8    osss_processor* m_processor;
9
10   Top() {
11     channel = new osss_rmi_channel<xilinx_opb_channel<false, false> >("channel");
12     channel->clock_port(Clk); channel->reset_port(Reset);
13     dct_inst = new osss_object_socket< osss_shared<dct> >();
14     dct_inst->clock_port(Clk); dct_inst->reset_port(Reset);
15     dct_inst->bind(*channel);
16     player1 = new player( "player1" );
17     player1->dct_port(*dct_inst);
18     m_processor = new xilinx_microblaze("my_processor");
19     m_processor->clock_port(Clk); m_processor->reset_port(Reset);
20     m_processor->rmi_client_port(*channel);
21     m_processor->add_sw_task(player1);
22   }
23 };
```

Listing 8.19: Top-Level module of the Virtual Target Architecture Layer model

The Software Task is executed on a Xilinx MicroBlaze processor and the Shared Object is implemented in hardware. For communication between the processor and the Shared Object a Xilinx OPB bus with a default data width of 32 bit has been chosen. Listing 8.19 shows the top-level module of the Virtual Target Architecture Layer model.

8.5.3.4 Implementation Model

Figure 8.24 shows a simplified block diagram of the FPGA platform configuration. The MicroBlaze processor is connected to a block RAM via the LMB interface for data (dlmb) and instructions (ilmb). This internal RAM contains the start-up code. The MP3 decoder software and its data structures are in an external DDR-SRAM. The MicroBlaze is connected to this external DDR-SRAM by multi-channel OPB DDR-SRAM controller (mch_opb_ddr) via data (dxcl) and instruction cache link (ixcl) interfaces. The DCT Shared Object with OPB interface is encapsulated in the `dct_top` block. A SystemACE controller (opb_sysace) is used to read an MP3 data stream from a Compact-Flash card. The AC97 digital controller (opb_ac97) is used to send the PCM samples for the left and right channel to a digital/analog converter and headphone amplifier. The hardware timer/counter module (opb_timer) is used for minimal intrusive execution time measurement on the FPGA prototype. The other components are used for debug purpose only.

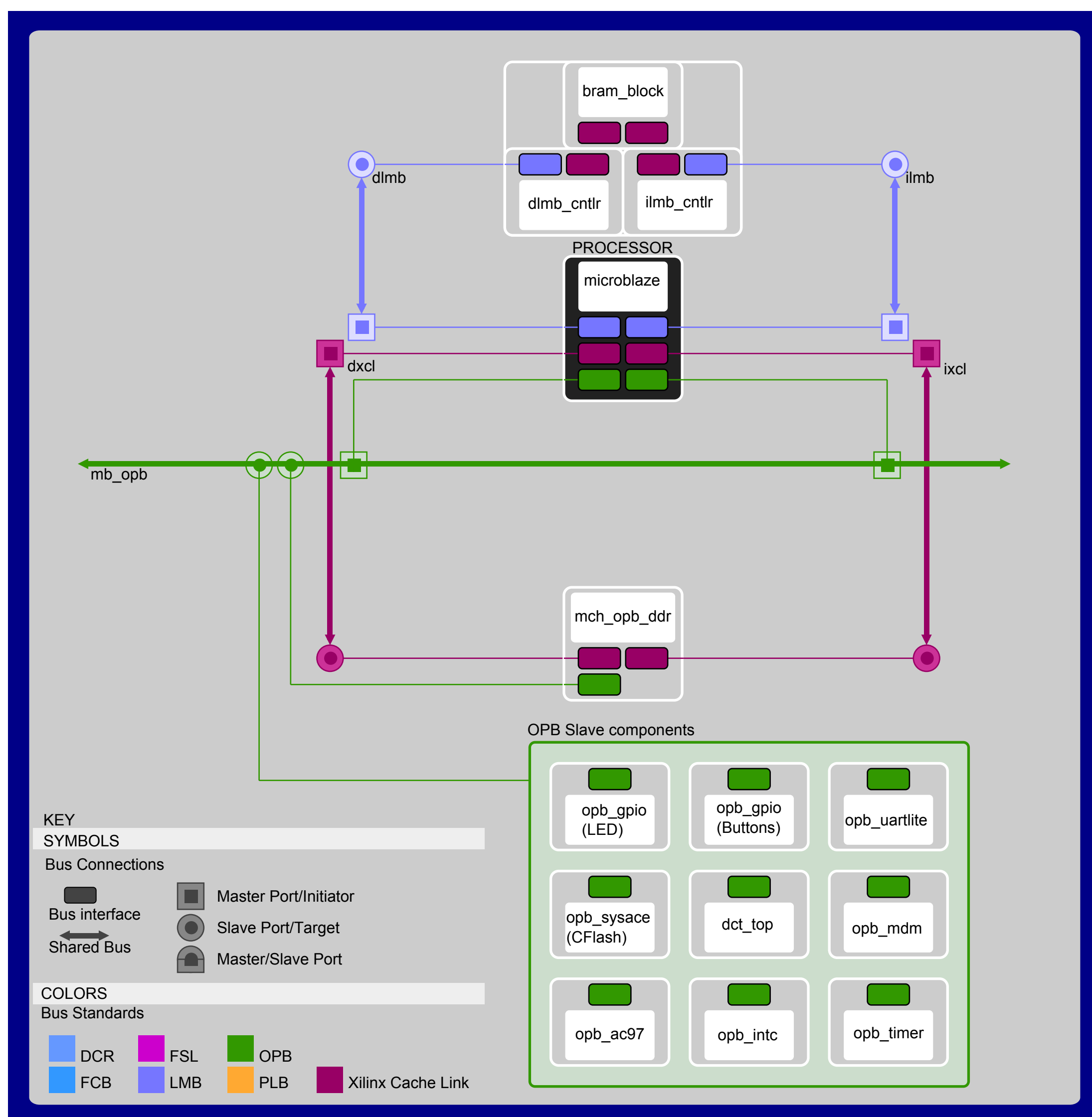


Figure 8.24: FPGA implementation platform configuration (simplified)

8.5.4 Results

All measurements have been performed on the implementation model using the hardware timer and MP3 data with a bit rate of 320 kb/s and a sampling frequency of 44,1 kHz. The chosen

bit rate is used for high quality MP3 streams and has the highest computation demand in the frequency domain. We have chosen the default sampling frequency, even though 48 kHz maximum sampling frequency would have resulted in a slightly higher number of DCT invocations per second. Anyhow, for the comparison of the pure software with the hardware/software implementation and a comparison of the RMI protocol and serialization overhead with a custom interface communication only a common MP3 stream is required.

8.5.4.1 Software implementation

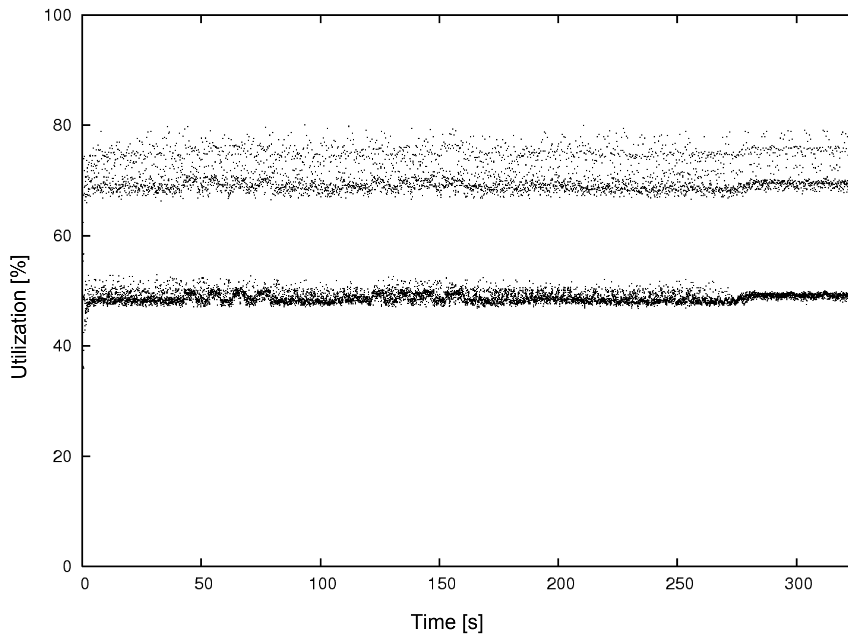


Figure 8.25: Processor utilization of the software only implementation [66]

Figure 8.25 shows a scatter plot of the utilization of the MicroBlaze processor for the software only implementation. Each data point represents the measured decoding time for one frame.The average processor utilization is 56,138% and the maximum utilization is 109,651%. The shown scatter plot has been recorded for a specific MP3 stream. Since the MP3 decoder has components with input data dependent execution times (e.g. Huffman Decoder), different MP3 streams will generate different processor utilizations.

For hiding the latency for the compact flash access, a ring buffer with configurable size is used. For this measurement, an input buffer size of 4.096 Byte has been chosen. With this buffer size three complete frames can be stored. The histogram in Figure 8.26 contains two groups or clusters. The first cluster contains frames that fit into the buffer without reloading data from the compact flash card. The second group represents frames where the buffer need to be refilled from the compact flash card. The ratio frames without buffer refill to frames with buffer refill is 2:1.

Further increasing the input buffer size enables real-time decoding of the software only implementation.

In addition to the overall frame decoding time, the execution time of the `dct32` function can be measured. For this purpose the hardware timer is read directly before and after each `dct32` call. This measurement has been performed for 12.500 frames and results in a minimal DCT execution time of 35,98 μs, a maximal execution time of 39,83 μs and an average execution time of 37,88 μs. This measurement is used for a comparison with the execution time of the hardware DCT and the RMI communication overhead.

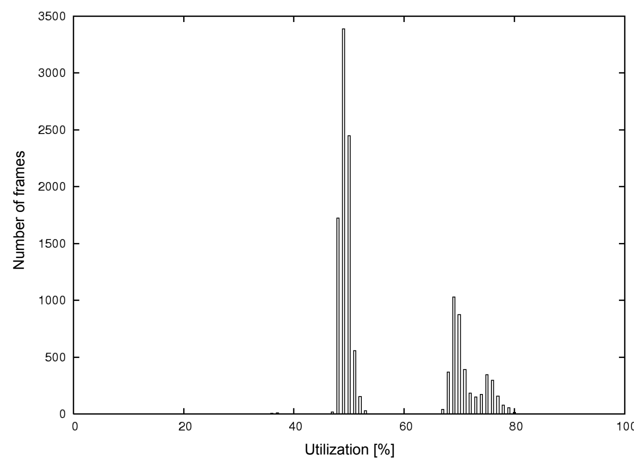


Figure 8.26: Histogram of the processor utilization from Figure 8.25 [66]

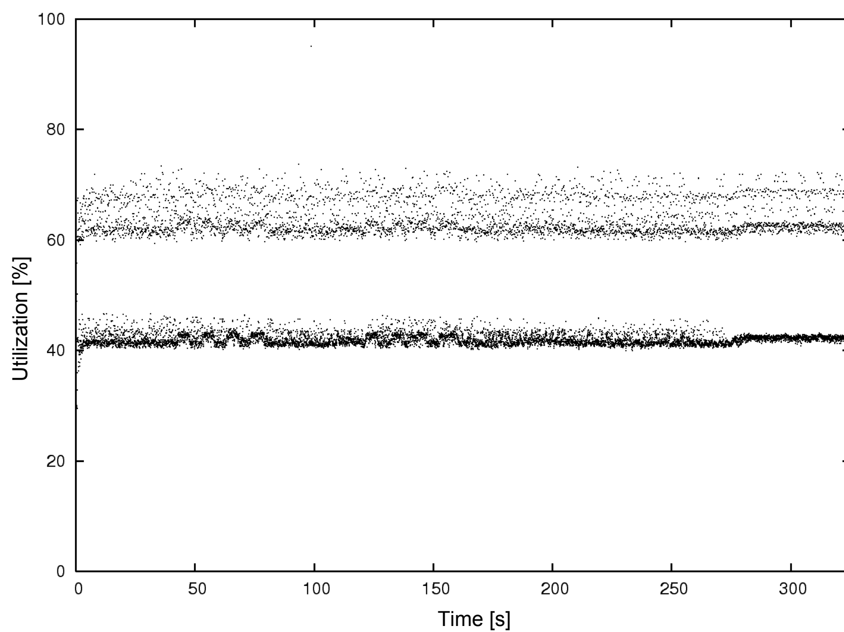


Figure 8.27: Processor utilization of the hardware/software implementation [66]

8.5.4.2 Hardware/Software implementation

Figure 8.27 shows a scatter plot and Figure 8.28 shows the corresponding histogram of processor utilization per frame for the decoding the same MP3 stream used above, but for an implementation with the DCT as dedicated hardware component. The utilization is lower, because the computation time of the hardware DCT is lower that the software DCT. The average utilization is 49,322% and the maximum utilization is 102,536%.

This hardware/software implementation does not take advantage of the parallel processing between hardware and software, e.g. through pipelining software and DCT. Thus, only an improvement of approximately 7% could be achieved. With a maximum utilization > 100% this implementation is still not capable for seamless real-time decoding. Doubling the input buffer from 10 KiB to 20 KiB results in a maximum utilization of 94% and an average utilization of 46%.

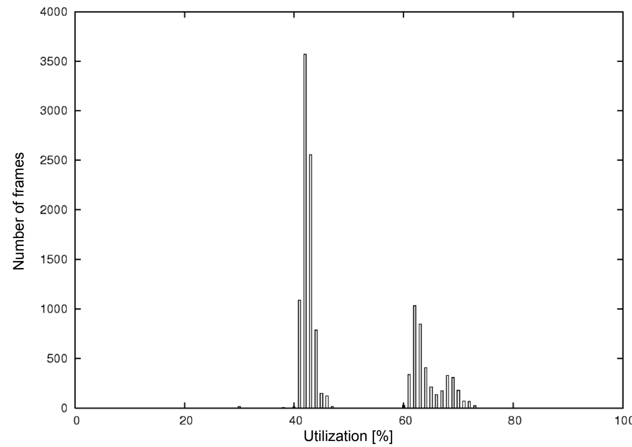


Figure 8.28: Histogram of the processor utilization from Figure 8.27 [66]

8.5.4.3 RMI overhead

For the evaluation of the communication overhead, the following measurements are necessary:

1. $\Delta(\mathrm{dct32})_{\mathrm{core}}$ is the pure DCT functional execution time in number of clock cycles.
2. $\Delta(\mathrm{dct32})_{\text{end-to-end}}$ is the time between starting the dct32 function and its completion in clock cycles.

Then, the communication time is:

$$\Delta(\mathrm{dct32})_{\mathrm{communication}} = \Delta(\mathrm{dct32})_{\text{end-to-end}} - \Delta(\mathrm{dct32})_{\mathrm{core}}$$

For a software implementation of the DCT:

$$\Delta(\mathrm{dct32})_{\mathrm{communication}} = 0 \Leftrightarrow \Delta(\mathrm{dct32})_{\text{end-to-end}} = \Delta(\mathrm{dct32})_{\mathrm{core}}$$

Table 8.7 gives a comparison of average DCT execution times $\Delta(\mathrm{dct32})_{\text{end-to-end}}$ for a pure software (SW) and a hardware/software (HW) implementation. For the comparison of the C++ and RMI overhead induced by OSSS, both implementations have been done in optimized C and C++ code, as used by OSSS. The C-based hardware/software implementation is not using the RMI protocol, but writes/reads the input/output `dct32` parameter array into the hardware DCT's register using memory mapped I/O. The C++ hardware/software implementation uses the RMI protocol and transfers the `dct_data` objects using the defined RMI protocol phases.

DCT impl.	software language	RMI	avg. exec. time [# clock cycles]	compared to SW DCT [%]
SW	C	no	3.788	100
HW	C	no	1.948	51,426
SW	C++	no	4.281	113,015
HW	C++	yes	2.339	61,748

Table 8.7: Average execution times of the different DCT implementations [66]

The usage of C++ in the software implementation extends the execution time of the DCT by 13%. The usage of C++ and RMI in the hardware/software implementation extents the execution time of the DCT by 20%. Thus, the overhead of the RMI protocol is 7% in this use-case.

Table 8.8 shows the different sizes of the executables. The optimized C implementation has the smallest size. The C++ implementation without RMI (i.e. without using the OSSS software

DCT impl.	SW language	RMI	size of executable [KiB]
HW	C	no	93,129
HW	C++	no	393,345
HW	C++	yes	1.755,055

Table 8.8: Size of executable overview for different implementations [66]

library) is more than 4 times bigger than the C implementation. This increase is mainly caused by the usage of the C++ libraries for standard input and output. The OSSS implementation is again more than 4 times bigger than the C++ implementation. This increase is caused by the full SystemC (synthesizable) data type support and the RMI library which makes use of the C++ Standard Template Library (STL). Since the OSSS software library has not been designed for memory size efficiency, optimizations to reduce the total memory size are still possible in the future.

8.5.5 Conclusion

This experiment has been designed to evaluate the communication overhead of OSSS compared to a handcrafted C implementation. The chosen MP3 decoder design is an optimized open-source C implementation and has been ported to the Xilinx MicroBlaze processor first. After initial software profiling, a simple non-pipelined hardware/software implementation, putting the most computation intensive part, i.e. the DCT of the subband synthesis, into dedicated hardware, has been implemented in C and VHDL and as OSSS Application and Virtual Target Architecture Layer models. The performance measurements on the FPGA target platform showed that the usage of C++ itself extends the execution time of the DCT by 13%. The OSSS hardware/software implementation extends the DCT execution time by 20%. The overhead of the RMI protocol is approximately 7%. This result shows, that the usage of C++ and RMI has a certain overhead to be considered and compared against the productivity gain.

8.6 IPv4 Packet Forwarding Switch

8.6.1 Goals of this experiment

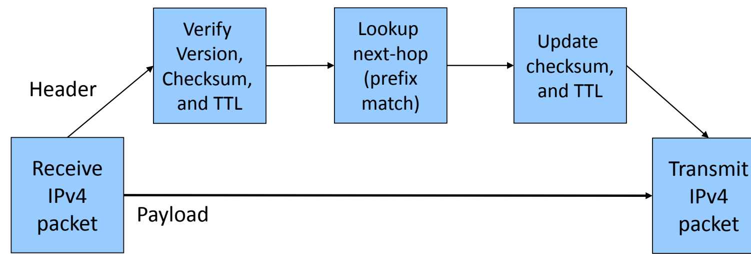
In this experiment a simple Internet Protocol (IP) packet forwarding switch, implemented in OSSS is used to evaluate the simulative design space exploration capabilities of the proposed OSSS methodology. This experiment is subdivided into two parts:

Simulation In the first part (see Section 8.6.3) different hardware/software partitionings on the Application Layer and different communication link to OSSS Channel mappings are analyzed for packet throughput using the OSSS simulation library. This part of the experiment has been previously published in [65].

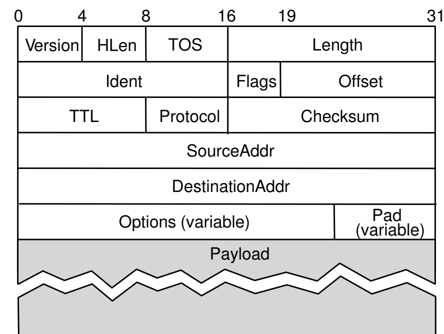
Synthesis In the second part (see Section 8.6.4) the automatic Shared Object synthesis of an IP Checker Shared Object is performed to demonstrate and evaluate

- the feasibility of the Shared Object synthesis approach,
- the chip (FPGA) area and throughput of Shared Objects depending on: the number of clients, interface type (point-to-point, bus) and channel bit width,
- the efficiency compared to a manual SystemC primitive channel refinement for hardware synthesis,
- and the provision of different communication configurations for design-space exploration without (re)design effort.

This part of the experiment has been previously published in [22].



(a) Block diagram of the router



(b) IPv4 packet structure

Figure 8.29: Overview of the IPv4 packet forwarding switch design

8.6.2 Introduction & Motivation

Figure 8.29a gives a general overview of the Internet Protocol (IP) packet switching algorithm. The IP header is split off from the IP packet because only the header contains information used for routing of the packet. Figure 8.29b shows the structure of IPv4 packets. For routing, the header's version, checksum and Time-to-live (TTL) fields are verified.

TTL is a value in an IP packet that tells a network router whether the packet has been in the network too long and should be discarded. For a number of reasons, packets may not be delivered to their destination in a reasonable length of time. For example, a combination of incorrect routing tables could cause a packet to loop endlessly. A solution is to discard the packet after a certain time and send a message to the originator, who can decide whether to resend the packet. The initial TTL value is set, usually by a system default, in an 8-binary digit field of the packet header. The original idea of TTL was that it would specify a certain time span in seconds that, when exhausted, would cause the packet to be discarded. Since each router is required to subtract at least one count from the TTL field, the count is usually used to mean the number of router hops the packet is allowed before it must be discarded. Each router that receives a packet subtracts one from the count in the TTL field.

For successful verified headers the next-hop is calculated and mapped to the corresponding output port of the router. In the last step, the TTL field is decremented by one and the header checksum is recalculated. IP header and Payload data are joined and sent out at the corresponding port of the router.

8.6.3 Modeling in OSSS

As a case study, we have implemented an IP version 4 router using the proposed OSSS methodology. Figure 8.31 shows the initial router design on the *Application Layer* together with its testbench. Figure 8.30 shows the class diagram of the implemented IP packet with support for serialization over OSSS RMI Channels.

The IP router itself consists out of six basic blocks: The Transceiver module receives and transmits IP packets from either the input or output FIFO queue. When receiving a packet its header is split off and sent to the Verify Shared Object. The Verify Shared Object checks whether the header is legal (e.g. checksum, time-to-life (TTL)). When this is the case, the

Transceiver stores the corresponding payload in the Payload Manager Shared Object. Otherwise, the entire packet is discarded. The Lookup module reads valid IP headers from the Verify Shared Object and performs a next-hop lookup on the Routing Table Shared Object (we have implemented the routing algorithm as described in [172]). Dependent on the lookup result the IP header is forwarded to either Update 0 or Update 1. The Update Shared Objects decrease the TTL field and update the header checksum. The corresponding Transceiver reads the header from the verifier, gets the corresponding payload from the Payload Manager, reassembles the IP packet and writes it to the output FIFO queue.

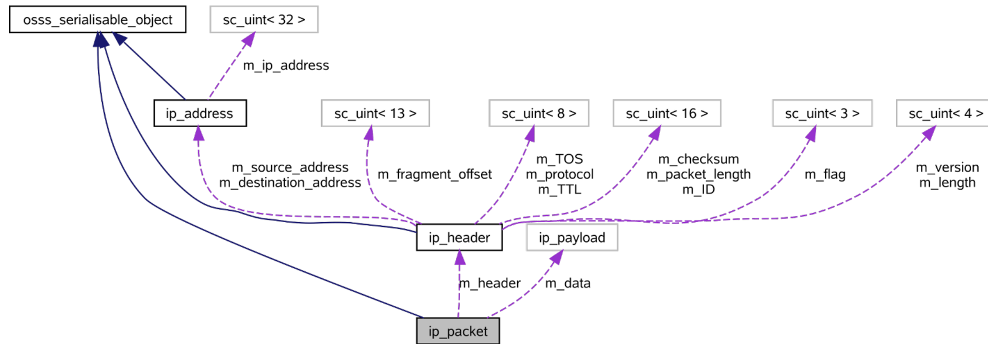


Figure 8.30: IP packet class diagram

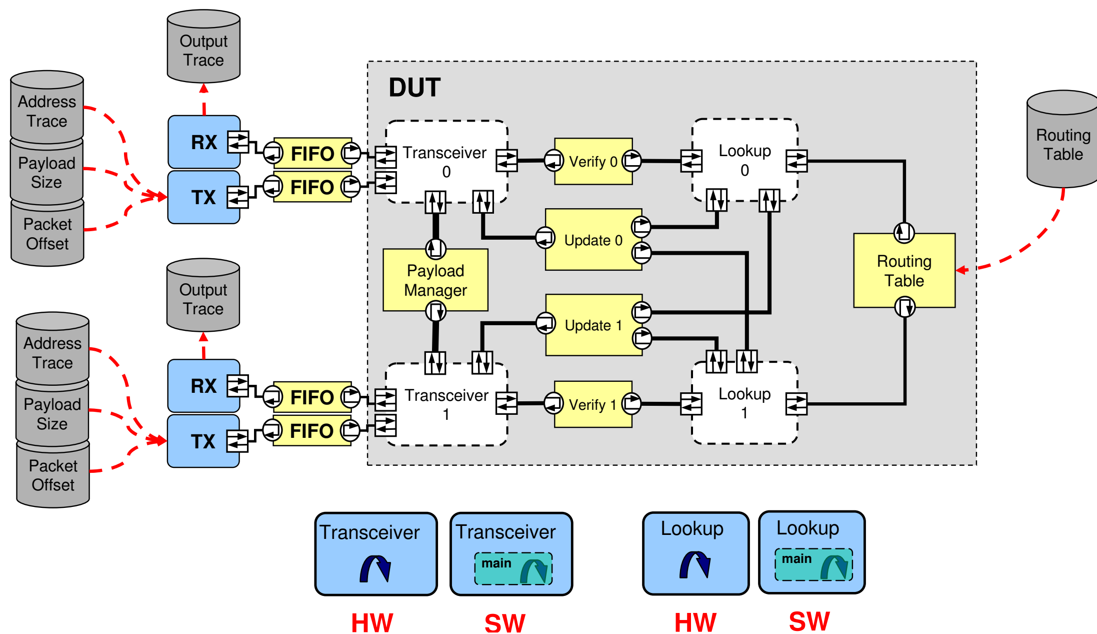
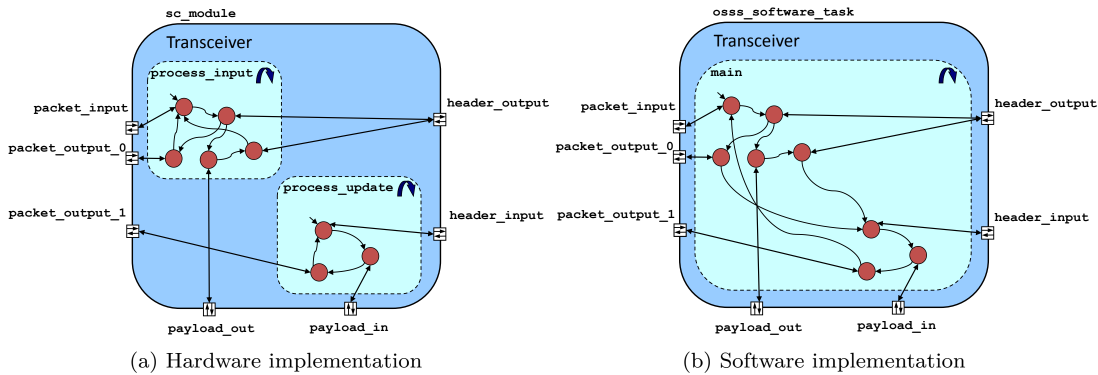


Figure 8.31: IPv4 router design with testbench on *Application Layer*

For simulation, we have instantiated IP sources (TX module) and sinks (RX module). The sources generate IP packets of a predefined address trace, payload size and packet offset (the time that passes between sending). The sink prints the routed packets to a file. For this example we have used a static routing table and address trace taken from [172]. The system clock used during simulation is 100 MHz.

The upper section of Table 8.9 (No. 1-4) shows the simulation results obtained on the *Application Layer* for different HW/SW partitionings. We have sent 10000 IP packets with increasing payload size and an offset of 30 clock cycles to both input FIFOs. The notation M_i with $M \in \{T, L\}, i \in \{\text{HW}, \text{SW}\}$ denotes whether the Transceiver (T) or the Lookup (L) module has been implemented as a hardware module (HW) or as a software task (SW).

As shown in Figure 8.31 at Application Layer different Transceiver and Lookup imple-



(a) Hardware implementation (b) Software implementation

Figure 8.32: Comparison of Transceiver Hardware and Software implementations

mentations can be plugged into the Design under Test (DUT). Figure 8.32 gives a graphical comparison between the structures of the Transceiver hardware and software implementations. In the hardware implementation model (Figure 8.32a) independent input and update processes are used to handle the incoming and outgoing IP packets. For the software implementation (Figure 8.32b) these two processes have been serialized into a pure sequential execution.

The experiments with different HW/SW partitionings on the *Application Layer* have shown that the average throughput is dominated by the Transceiver block. The more blocks are implemented in SW the lower the average packet throughput gets. The highest achievable average throughput has been highlighted in Table 8.9[1].

The two lower parts of Table 8.9 (No. 5-10) show the results after the mapping to the *Virtual Target Architecture Layer* has been performed. Software Tasks have been mapped to Xilinx MicroBlaze processors and communication links have been mapped either onto dedicated point-to-point or Xilinx OPB channels. The notation $M_i \xrightarrow{c} *$ denotes that all communication links originating from module $M \in \{T, L\}$ implemented in $i \in \{\text{HW}, \text{SW}\}$ have been mapped on channel $c \in \{\text{p2p}, \text{OPB}\}$. U instead of $*$ denotes that only the connection to the Update Shared Objects has been mapped onto channel c. Due to the increased latency and simulation time, we have changed the number of input IP packets to 2000 and the offset to 500 and 1500 clock cycles. Figure 8.33 shows two different Virtual Target Architecture implementations of the IPv4 design.

The experiments on the *Virtual Target Architecture Layer* with more accurate throughput results have confirmed the observations made on the *Application Layer*.

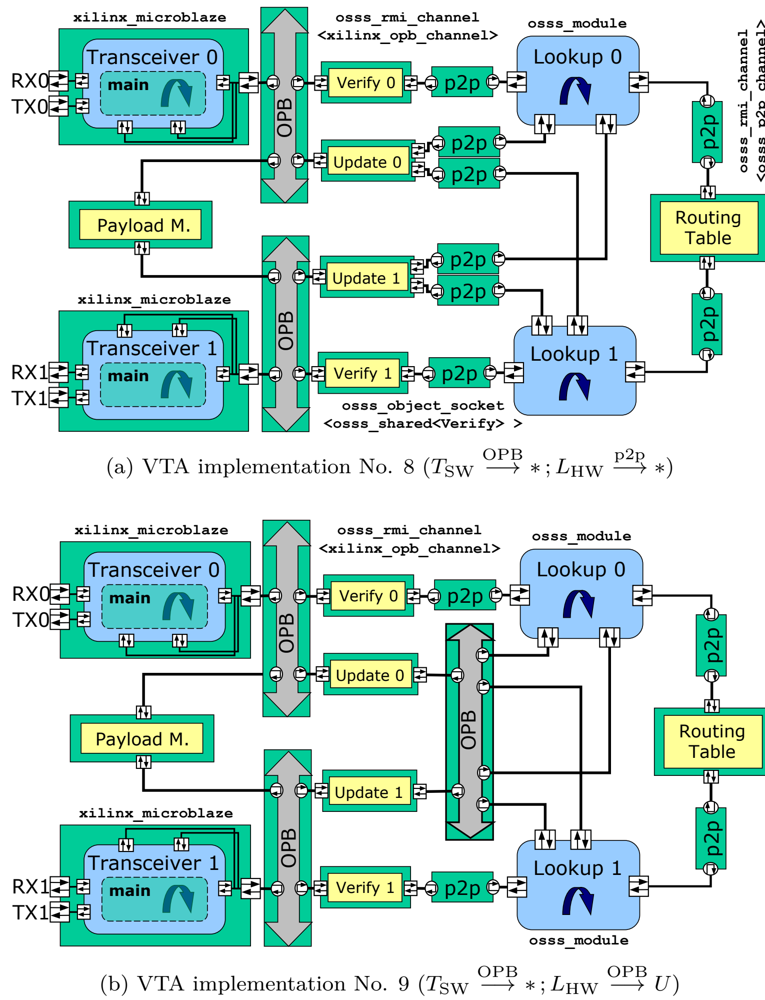
In the first part of this experiment, we have presented OSSS as a homogeneous, object-oriented, executable and synthesizable modeling language for embedded HW/SW systems on different levels of abstraction. The *Application Layer* provides a functional view of the system. The designer can explore the amount of available parallelism and make first a first HW/SW partitioning decision. On the *Virtual Target Architecture Layer* software tasks and communication links are mapped on processors and bit-accurate synthesizable channels. By changing the interconnection network on the Virtual Target Architecture Layer the impact of different communication resources on the data throughput can be evaluated through simulation. Figure 8.34 gives a summary of the IPv4 design and architecture exploration phase of the first part of this experiment.

8.6.4 Synthesis

The objectives of the synthesis experiment presented in this section are to demonstrate

- the feasibility of the Shared Object synthesis approach,
- the efficiency compared to a manual SystemC primitive channel refinement for hardware synthesis,

[1] As one can see the transceiver dominates the throughput performance. We have chosen this implementation to be the best, because it is expected to be the most cost-efficient.



(a) VTA implementation No. 8 ($T_{\mathrm{SW}} \stackrel{\mathrm{OPB}}{\longrightarrow} *; L_{\mathrm{HW}} \stackrel{\mathrm{p2p}}{\longrightarrow} *$)

(b) VTA implementation No. 9 ($T_{\mathrm{SW}} \stackrel{\mathrm{OPB}}{\longrightarrow} *; L_{\mathrm{HW}} \stackrel{\mathrm{OPB}}{\longrightarrow} U$)

Figure 8.33: Two different Virtual Target Architecture implementations of the IPv4 design

- and the provision of different communication configurations for design-space exploration without (re)design effort.

To demonstrate the feasibility of the approach, Shared Object synthesis results of the packet processing part taken from the HW Internet Protocol (IP) packet forwarding switch, as described above, are presented. Figure 8.35 gives an overview of the IPv4 design Shared Object synthesis experiment.

The design consists of a single producer process, an IP checker Shared Object, and a variable number of packet consuming processes. The IP checker Shared Object is connected to its client processes through a bit-width scalable point-to-point channel. The detailed structure and the parameters of the design are shown in Figure 8.36.

The producer process generates a continuous stream of IP packets. More specifically, one IP packet each clock cycle, but it stops the production of more packets when the buffer space in the IP checker Shared Object is full. It continues with the packet production in the case of available buffer space.

The IP checker performs IP header version, checksum, and time-to-live (TTL) validation upon reception (`put` method) and accepts the packet for delivery (`get` method) only if the IP header version and checksum are correct, and the TTL value is bigger than zero. For delivery, the TTL field is decremented and the checksum is updated. Listing 8.20 shows the IP packet header. Omitting the optional options and padding filed of the IP header, the total size of the header is 160 bit. Listing 8.21 shows the implementation of the IP checker Shared Object.

No.			∅ **Throughput** [byte/s]	**Simulation[a] Time** [s]
Application Layer (Nr. packets: 10000, offset: 30)				
1	T_{HW}	L_{HW}	7.44047e+10	848.74
2	T_{HW}	L_{SW}	**7.44047e+10**	864.61
3	T_{SW}	L_{HW}	4.74822e+10	629.20
4	T_{SW}	L_{SW}	4.70995e+10	710.03
Virtual Target Architecture Layer (Nr. packets: 2000, offset: 500)				
5	$T_{\text{HW}} \xrightarrow{\text{p2p}} *$	$L_{\text{HW}} \xrightarrow{\text{p2p}} *$	1.33018e+08	1500.54
6	$T_{\text{HW}} \xrightarrow{\text{p2p}} *$	$L_{\text{HW}} \xrightarrow{\text{OPB}} U$	1.33020e+08	1483.57
7	$T_{\text{HW}} \xrightarrow{\text{p2p}} *$	$L_{\text{SW}} \xrightarrow{\text{OPB}} *$	**1.33020e+08**	1494.96
Virtual Target Architecture Layer (Nr. packets: 2000, offset: 1500)				
8	$T_{\text{SW}} \xrightarrow{\text{OPB}} *$	$L_{\text{HW}} \xrightarrow{\text{p2p}} *$	4.1988e+07	11899.48
9	$T_{\text{SW}} \xrightarrow{\text{OPB}} *$	$L_{\text{HW}} \xrightarrow{\text{OPB}} U$	4.20569e+07	11066.92
10	$T_{\text{SW}} \xrightarrow{\text{OPB}} *$	$L_{\text{SW}} \xrightarrow{\text{OPB}} *$	4.19786e+07	11892.36

[a] Intel(R) Pentium(R) 4 @3.00GHz, 1MB Cache, 1GB RAM

Table 8.9: Simulation results of different IPv4 router implementations [65]

The IP packets are received by a scalable number of consumer processes that represent a set of uniform processing elements. This example design can be thought of as a converter that inspects a serial stream of IP packets, and distributes them over multiple parallel data links to other processing units.

For comparing the efficiency of the proposed Shared Object synthesis approach with a manual SystemC primitive channel refinement, we have performed a stepwise refinement of a functional equivalent pure SystemC high-level model down to a HW synthesizable SystemC model. To quantify the effort of both implementations we consider the lines of code (LOC) for both implementation paths.

Design Entity	**Manual Artifact**	**OSSS** [LOC]	**SystemC** [LOC]
IP packet		34	34
producer		28	28
consumer		41	41
channel interface		10	10
channel behavior		27	40
	refined behavior	-	153
	access scheduler	-	162
	RMI protocol	-	113
top-level design		37	35
physical channel		10	-
	client interface	-	80
	server interface	-	129
	P2P channel	10	64
Total		197	889

Table 8.10: Comparison of automatic OSSS vs. manual SystemC implementation [22]

Table 8.10 shows the experimental results of the design efficiency study based on the IP checker design. In the first column it shows the main entities of the design: producer module, consumer module, channel interface definition, channel behavior/interface implementation, and top-level design. These entities are present in both, the pure SystemC and the OSSS design. The second column shows the main artifacts of the manual refinement of the pure SystemC

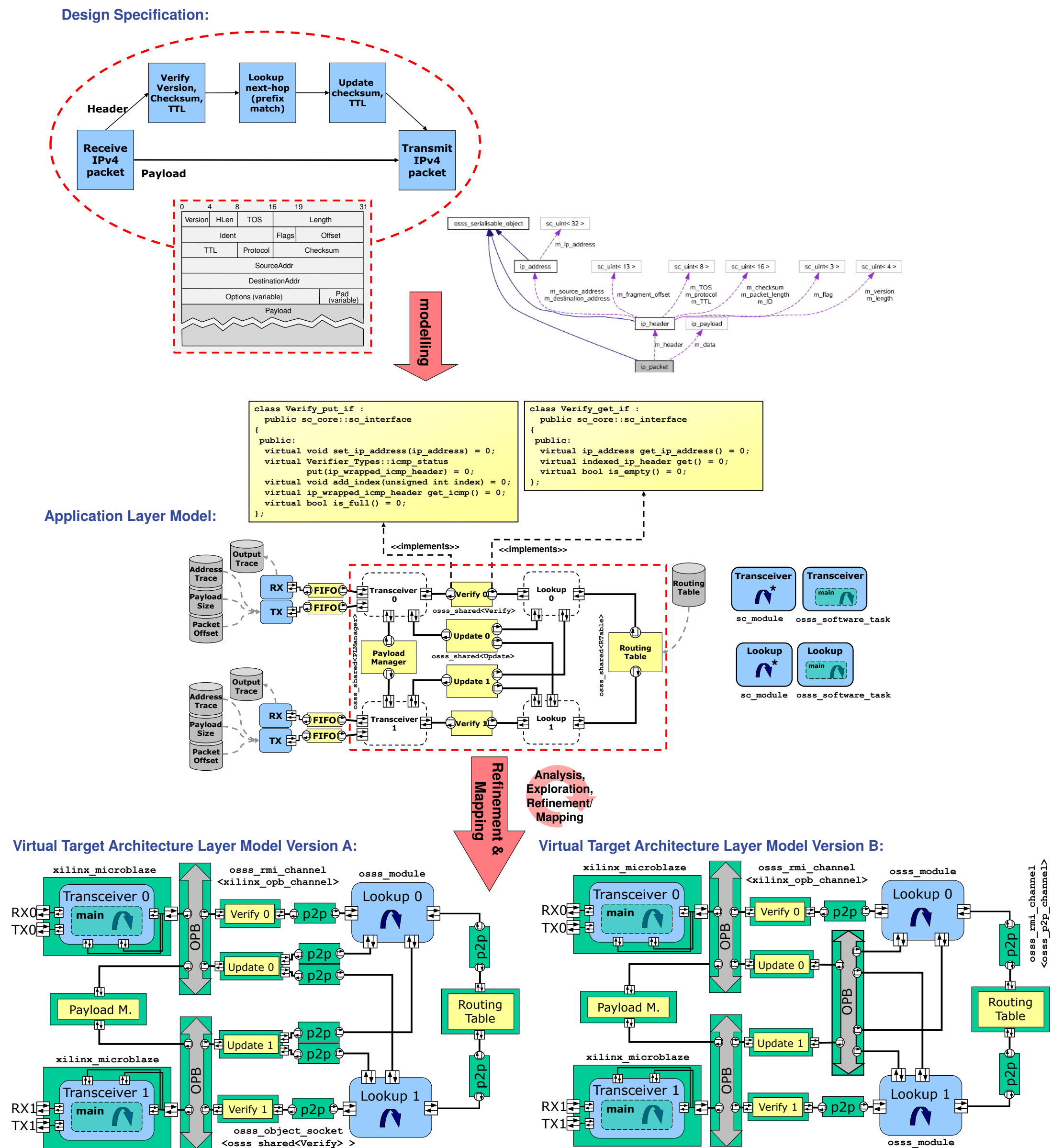


Figure 8.34: Summary of the IPv4 design and architecture exploration

design towards a synthesizable HW implementation. These artifacts are: the refinement of the channel's behavior to be integrated with the access scheduler and the RMI protocol, the client and server interfaces to the point-to-point (P2P) protocol, and the P2P channel itself. The presented LOC metric only gives a rough impression of the manual refinement effort. In our simple example we have measured a factor four gain in productivity. All manual refinement steps of the pure SystemC design are indeed error-prone and time consuming. Moreover, the maintenance and the scalability of the number of connected client processes, different scheduling policies, and P2P channel communication bit-widths is non-trivial compared to the automatic Shared Object synthesis and adds a big extra effort.

For this purpose we have developed the synthesis tool *Fossy* [225], that is able to perform these time-consuming and error-prone Shared Object and communication refinement steps automatically. By assigning different parameters to the OSSS input model we can perform complex design-space exploration (DSE) for different communication implementations, number of client processes, scheduling policies, and physical channel implementations. In the next experiment we will show, that finding the right balance between the number of packet consuming

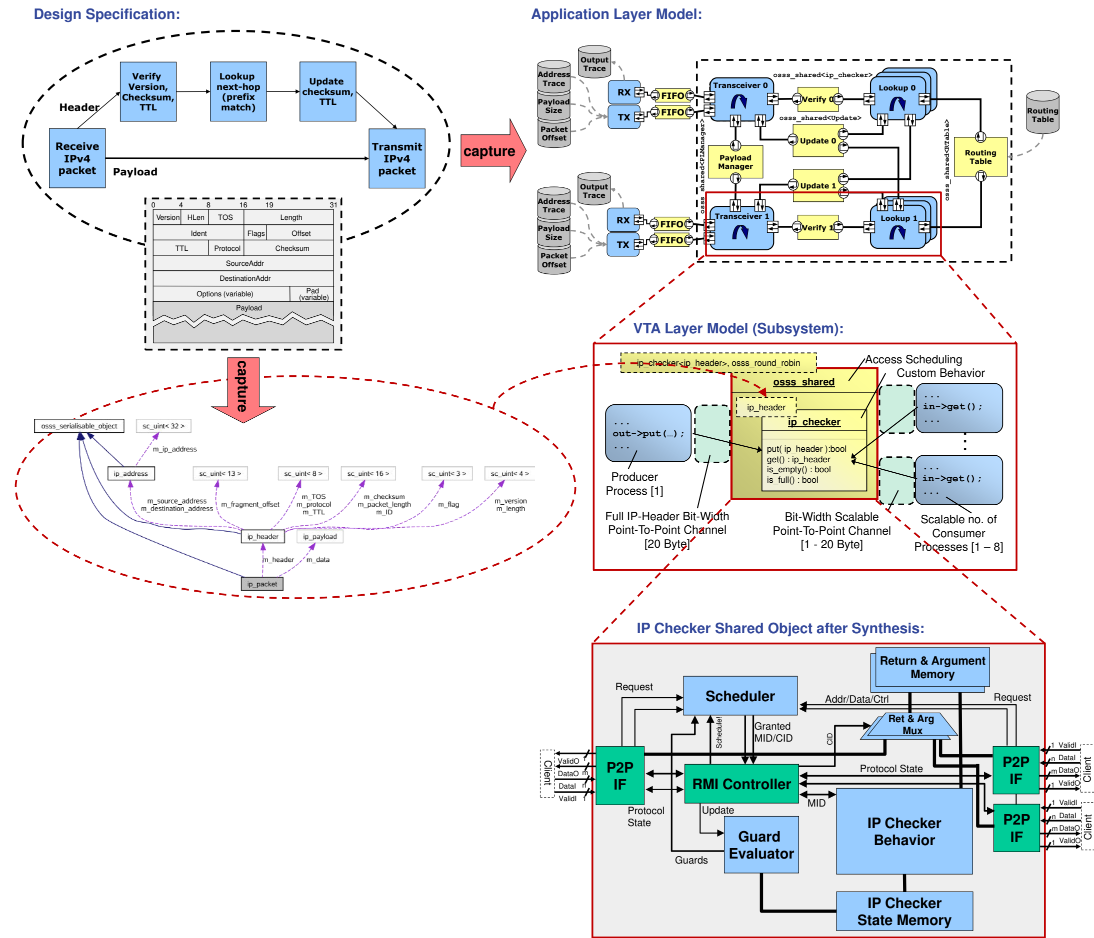


Figure 8.35: Overview of the IPv4 design Shared Object synthesis experiment

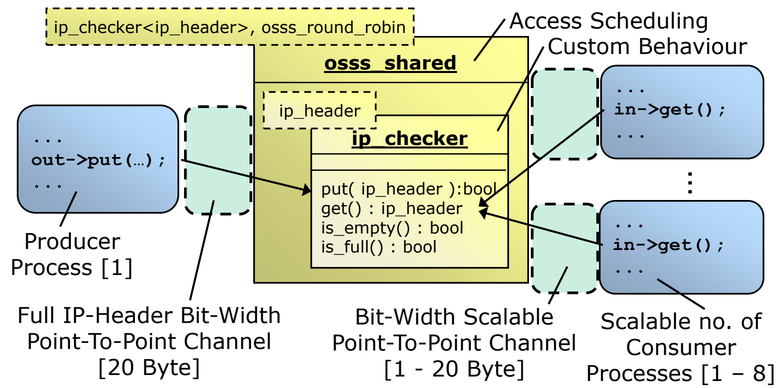


Figure 8.36: Structure and configuration parameters of the IP-Checker design [22]

processes, packet throughput, and area consumption is a non-trivial task, even in this simple IP checker design. Using our methodology and synthesis tool enables a DSE without any manual refinement or (re)design effort.

channel width [bit]	**throughput [packets/s]**	**# LUTs**	**# FFs**	f_{max} **[MHz]**
number of consumers: 1				
160 (full)	2,387,000	386	360	154
64	2,280,000	548	702	152
32	2,204,000	719	852	152
8	1,848,000	1145	1142	154
4	1,463,000	1388	1083	154
number of consumers: 2				
160 (full)	**4,117,500**	**522**	**455**	135
64	4,248,000	692	838	144
32	3,876,000	934	1112	136
8	3,172,500	1743	1636	135
4	2,145,000	1889	1585	130
number of consumers: 4				
160 (full)	**4,650,000**	873	631	93
64	4,095,000	1079	1135	90
32	3,818,000	1462	1563	92
8	2,397,000	2832	2618	94
4	1,518,000	3083	2573	92
number of consumers: 6				
160 (full)	3,000,000	1201	819	60
64	2,821,000	1629	1446	62
32	2,531,500	1996	2061	61
8	1,479,000	4085	3604	58
4	1,006,500	4476	3432	61
number of consumers: 8				
160 (full)	2,700,000	2211	914	54
64	2,457,000	2704	1713	54
32	2,116,500	3142	2468	51
8	1,351,500	5905	4545	53
4	874,500	6394	4529	53

Table 8.11: Results of the automatic HW synthesis experiments [22]

In this DSE experiment, we evaluated several combinations of channel bit-widths and numbers of consumer processes, as shown in Table 8.11. The results of Table 8.11 are visualised as surface plot in Figure 8.37 and Figure 8.38. The figures were gained by using our synthesis tool and processing its generated VHDL output with the Xilinx Synthesis Tool (XST) [238], targeting a Xilinx Virtex 4 LX25 [91] FPGA. The performance figures were obtained through post-synthesis simulation, driving the design with f_{max}[2] as clock frequency.

The columns of Table 8.11 show the chosen bit width of the consumer process' channels, the throughput (i.e. the number of packets processed per second), as well as the number of utilized HW resources (LUTs and register-bits), and the maximum achievable clock frequency. The experiments have been repeated for a different number of connected consumer processes.

The results show that increasing the number of consumers does not necessarily increase the throughput. This is because adding more consumers will not increase the throughput until the

[2] f_{max} is the maximum clock frequency reported by XST after synthesis, mapping and place and route (critical path) static timing analysis.

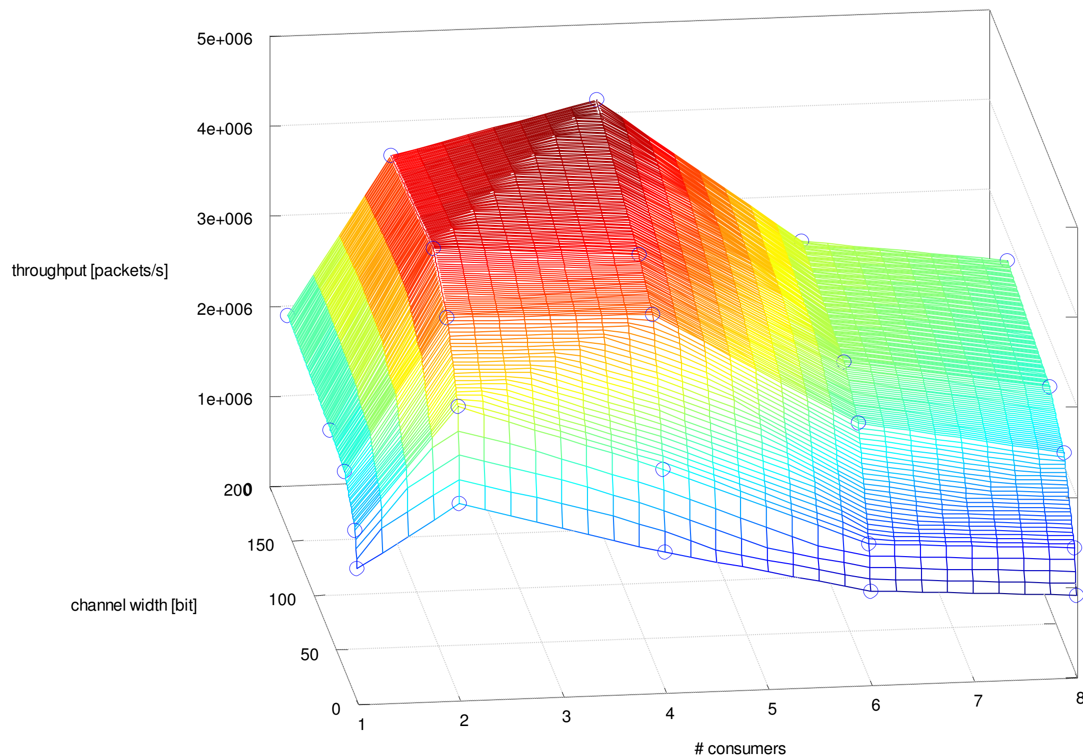


Figure 8.37: Visualization of throughput [packets/s] from Table 8.11

point where the time between two requests by the same consumer is lower than the time it takes to serve the requests of all other consumers. Such a scenario will leave some of the consumers in a waiting state at all times. Adding more consumers beyond this limit has the opposite effect since the increase in area consumption decreases the maximum clock frequency and thus reduces the overall throughput.

Since the bit-width of the channel directly affects the time it takes to transmit one data packet, the total time taken to serve a request by a client is also affected. This entails that the optimum number of clients shifts towards lower values for channels with a small bit-widths, as can be seen when comparing the throughput values for the designs with two, respectively four consumers.

Furthermore, our results indicate that the lowest resource utilization can be achieved by using the full width of the data packet as bit-width for the communication channel. Due to the inevitable rise in complexity of the communication interfaces inside the packet buffer where data transfers have to be split into several chunks.

This small synthesis experiments demonstrates the usefulness of the automatic synthesis approach for DSE. It assists the designer in finding the sweet spots in custom communication intensive HW designs.

```
1  namespace ip_header_types {
2    enum protocol {
3      ICMP = 1, // Internet Control Message Protocol
4      TCP =  6, // Transmission Control Protocol
5      UDP = 17  // User Datagram Protocol
6    };
7  }
8
9  class ip_header {
10  public:
11   ip_header();
12   ip_header(const ip_header& iph);
13   ip_header& operator=(const ip_header& iph);
14   bool operator==(const ip_header&) const;
```

```
15
16   //data field setters and getters
17
18   void version(const sc_uint<4>& v);
19   const sc_uint<4>& version() const;
20
21   const sc_uint<4>& length() const;
22
23   void TOS(const sc_uint<8>& v);
24   const sc_uint<8>& TOS() const;
25
26   void packet_length(const sc_uint<16>& v);
27   const sc_uint<16>& packet_length() const;
28
29   void ID(const sc_uint<16>& v);
30   const sc_uint<16>& ID() const;
31
32   void flag(const sc_uint<3>& v);
33   const sc_uint<3>& flag() const;
34
35   void fragment_offset(const sc_uint<13>& v);
36   const sc_uint<13>& fragment_offset() const;
37
38   void TTL(const sc_uint<8>& v);
39   const sc_uint<8>& TTL() const;
40   bool decrement_TTL();
41
42   void protocol(const ip_header_types::protocol& prot);
43   ip_header_types::protocol protocol() const;
44
45   void source_address(const ip_address& v);
46   const ip_address& source_address() const;
47
48   void destination_address(const ip_address& v);
49   const ip_address& destination_address() const;
50
51   bool checksum_valid();  // return true when header checksum is valid
52   void update_checksum(); // calculates new checksum
53   void decrement_ttl();   // decrements ttl field (without explicitly updating checksum)
54
55  protected:
56   sc_uint<4>  m_version;
57   sc_uint<4>  m_length;
58   sc_uint<8>  m_TOS;
59   sc_uint<16> m_packet_length;
60   sc_uint<16> m_ID;
61   sc_uint<3>  m_flag;
62   sc_uint<13> m_fragment_offset;
63   sc_uint<8>  m_TTL;
64   sc_uint<8>  m_protocol;
65   sc_uint<16> m_checksum;
66   ip_address  m_source_address;
67   ip_address  m_destination_address;
68 };
```

Listing 8.20: IP header class used in the experiment

```
1  class put_if : public virtual sc_interface {
2  public:
3    virtual bool put(ip_header iph) = 0;
4  };
5
6  class get_if : public virtual sc_interface {
7  public:
8    virtual ip_header get() = 0;
9  };
10
11 class ip_checker : public put_if, public get_if {
12 public:
13   ip_checker() : m_full(false), m_empty(true), m_data() {}
14
```

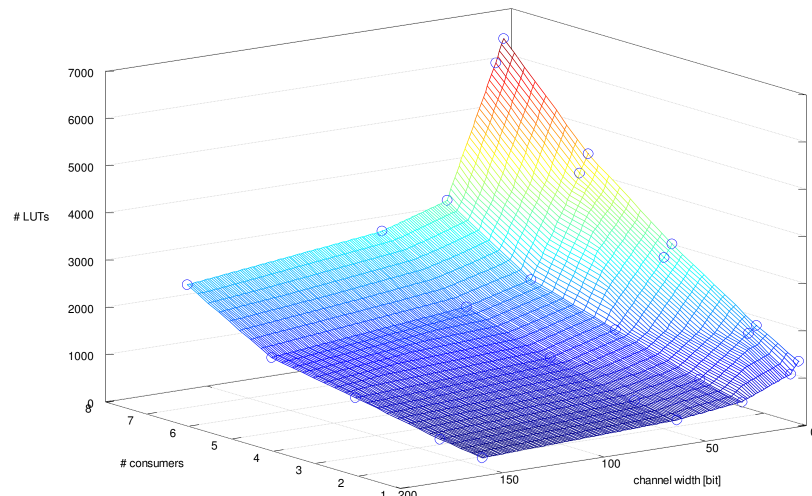
```
15   OSSS_GUARDED_METHOD(bool, put, OSSS_PARAMS(1, ip_header, iph), !m_full) {
16     if (iph.version().to_uint() != 4) return false;
17     bool csum_ok = iph.checksum_valid();
18     bool ttl_ok  = iph.TTL().to_uint() == 0 ? false : true;
19     if (csum_ok && ttl_ok) {
20     m_data = iph;
21       m_empty = false;
22       m_full = true;
23     }
24     return (csum_ok && ttl_ok);
25   }
26
27   OSSS_GUARDED_METHOD(ip_header, get, OSSS_PARAMS(0), !m_empty) {
28     m_empty = true;
29     m_full = false;
30     m_data.decrement_ttl();
31     m_data.update_checksum();
32     return m_data;
33   }
34
35   private:
36     bool  m_full, m_empty;
37     ip_header m_data;
38 };
```

Listing 8.21: IP checker class used in the experiment

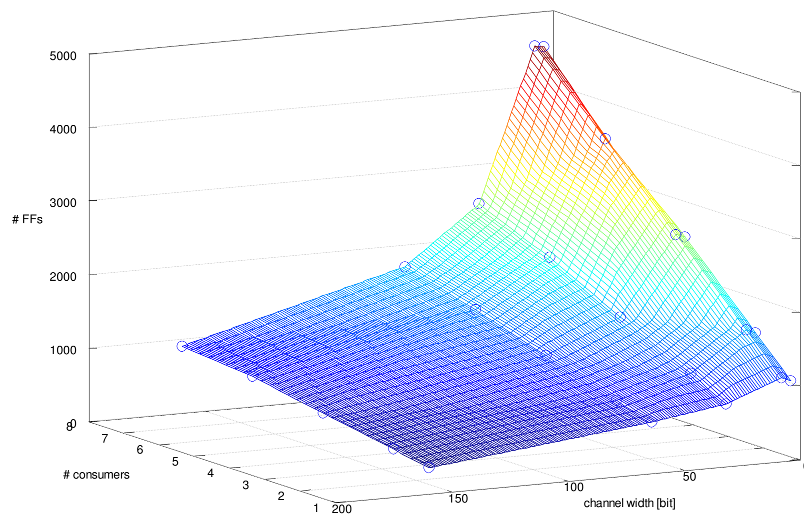
8.6.5 Conclusion

In the first part of this experiment, we have demonstrated the interchangeability of hardware and software implementations at the Application Layer. Furthermore, the influence on the packet throughput of different hardware/software configurations and communication link to OSSS Channels (point-to-point and shared bus) has been analyzed through simulation.

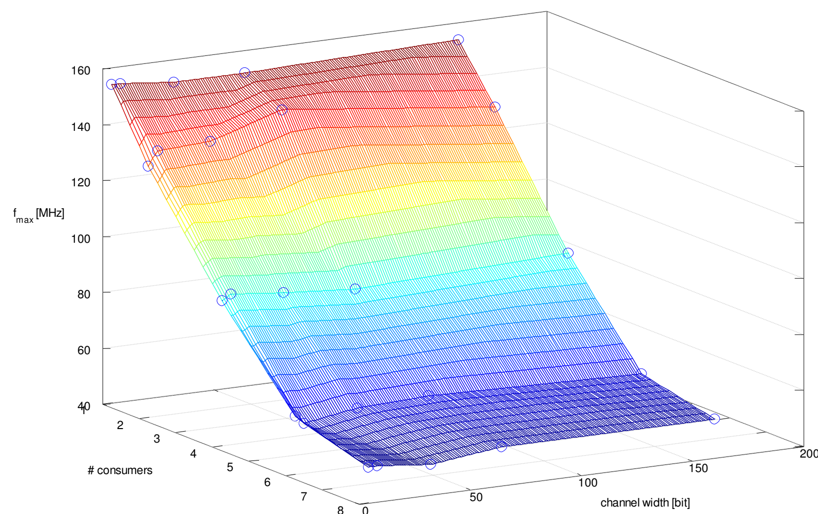
In the second part, we have demonstrated the feasibility of our approach through synthesis of a communication intensive packet processing design. However, the synthesis results have shown that the maximum throughput cannot be achieved by simply scaling up the number of clients. Even though, enough communication headroom would be available. These restrictions depend on the resulting complexity of the Shared Object which directly affects the critical path, and thus f_{max}.



(a) number of LUTs



(b) number of FF



(c) f_{max} [MHz]

Figure 8.38: Visualization of # LUTs, # FFs, and f_{max} [MHz] from Table 8.11

8.7 JPEG 2000 Decoder

8.7.1 Goals of this experiment

This experiment describes and evaluates the OSSS methodology for embedded hardware/software systems and its use in a JPEG 2000 decoder case study. The goal of this experiment is to identify the most promising parallel structure by comparing different design alternatives on the Application Layer. Furthermore, the experiment demonstrates usage of the OSSS refinement process, using the Virtual Target Architecture Layer, for analysis of the system behavior at cycle-accurate granularity and support of a simulative exploration of different target architectures for the JPEG 2000 decoder. Finally, this experiment quantitatively compares the OSSS custom hardware synthesis approach with a standard design approach using an industrial C++/VHDL-based implementation of the JPEG 2000 decoder on a Xilinx Virtex-4 FPGA.

This experiment has been previously published in [62] and [45].

8.7.2 Introduction

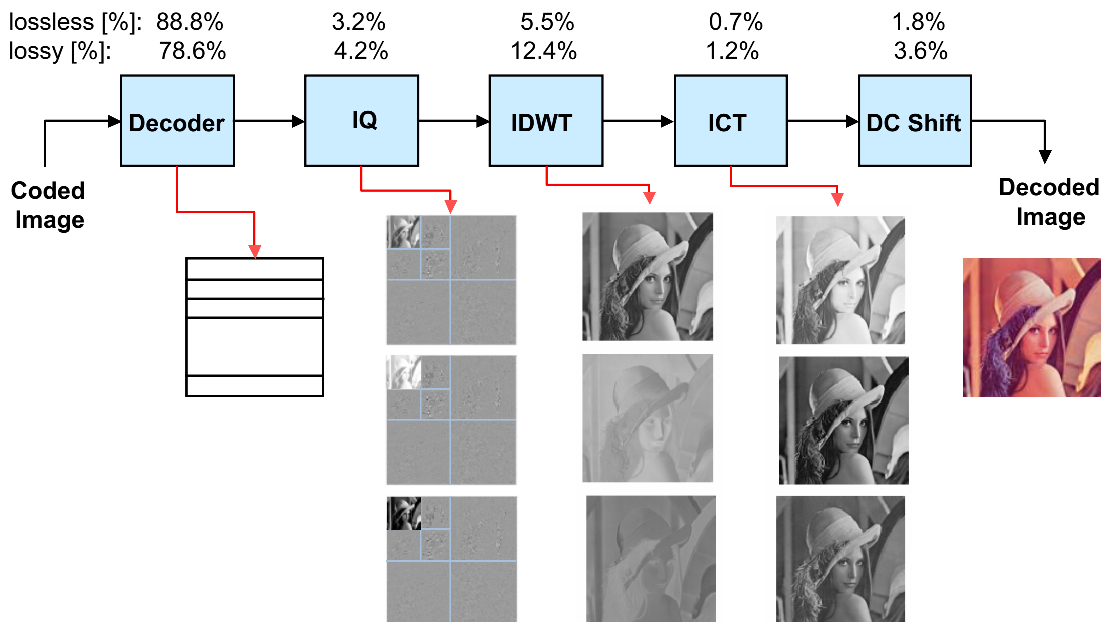


Figure 8.39: JPEG 2000 processing overview [45]

The JPEG 2000 decoder used in this experiment is a recent compression standard targeting different types of still images (bi-level, gray-level, color, multi-components, etc.). It supports different characteristics (natural images, scientific, medical, remote sensing imagery, text, etc.) allowing different imaging models (client/server, real-time transmission, image library archival, etc.) within a unified system.

As shown in Figure 8.39 the JPEG 2000 decoding chain is composed of several functional blocks, each of which performs a specific part of the image processing. The core components are the arithmetic decoder and the IDWT (**I**nverse **D**iscrete **W**avelet **T**ransformation). These two steps allow compressing and organizing data efficiently. In our case-study the JPEG 2000 decoder supports a lossy (IDWT 97) and lossless (IDWT 53) mode. Most JPEG 2000 images are processed as tiles (small parts of the image), which are more manageable and more adapted to a pipelined computation. For more information about JPEG 2000, please refer to [144].

8.7.3 Modeling in OSSS

8.7.3.1 Application Layer Model

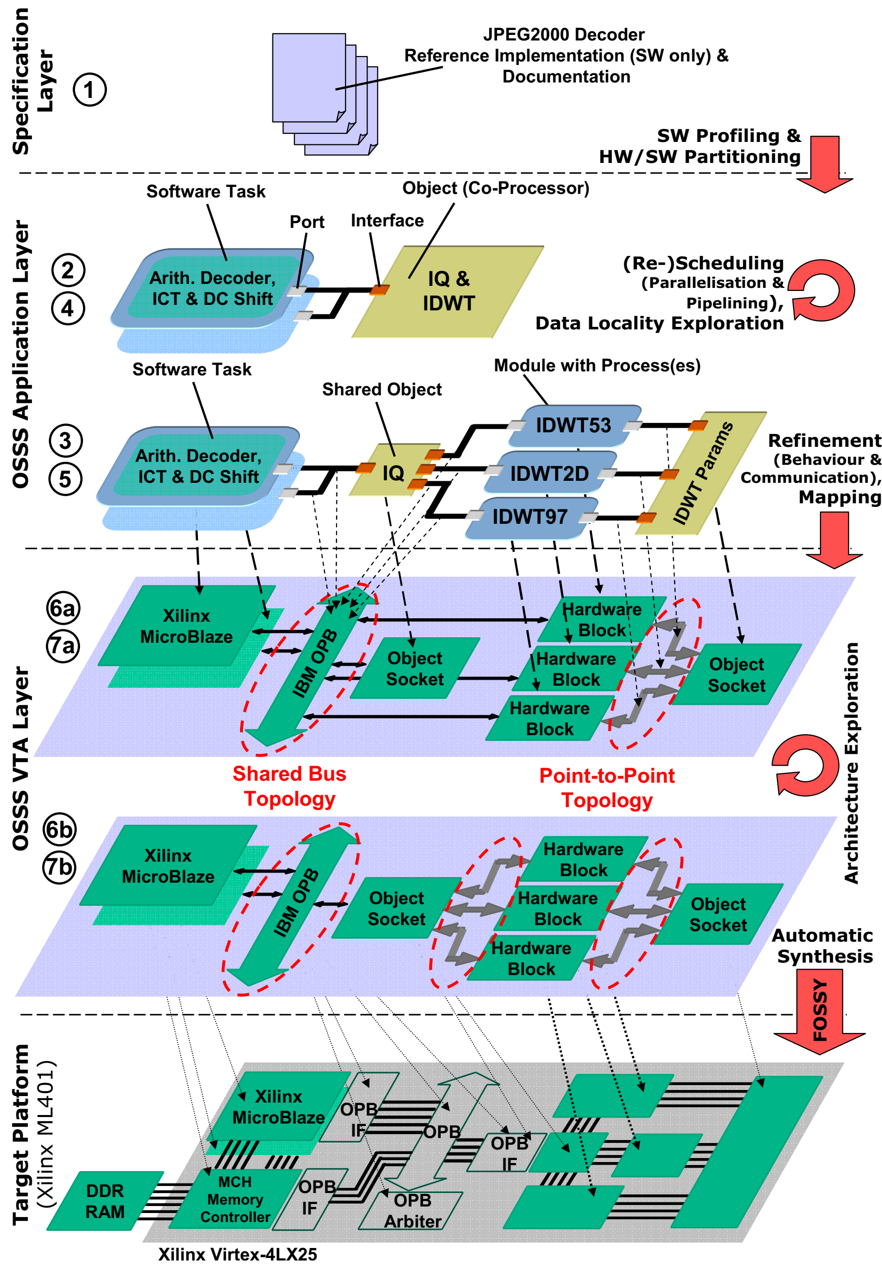


Figure 8.40: Implementation flow [45]

Figure 8.40 shows the Implementation flow of the JPEG 2000 decoder design. The starting point for our case-study was an existing C/C++ implementation of a JPEG 2000 decoder ①.

A profiling run of the JPEG 2000 decoder on a Xilinx MicroBlaze™ processor (because it is the only available target processor considered in this case study) points out the execution times of the algorithmic parts, shown in percentage of the overall computation (for the lossless and the lossy version) in the upper two lines of Figure 8.39.

The analysis of a software profiling taking into account the number of calls and the execution time of the algorithmic parts, we decided to parallelize the arithmetic decoder (88.8% for lossless, 78.6% for lossy) and to relocate the IDWT (5.5% for lossless, 12.5% for lossy) and inverse quantization (IQ, 3.2% for lossless, 4.2% for lossy) to hardware.

In the initial Application Model a *Software Task* contains the arithmetic decoder, ICT and DC Shift (normalisation) ②. The *Shared Object* (SO), which here serves merely as a co-processing unit, implements the IQ and IDWT. The software performs first the decoding, calls the IQ, calls the IDWT and finally continues with the ICT and the DC Shift. Since all method-calls to Shared Objects are blocking, the software execution cannot proceed until the SO has finished its computation. The simulation results in Table 8.12 show a speed-up of about 10% for lossless and 19% for lossy decoding compared to ①. This speed-up is higher than the maximum obtainable speed-up because it does not incorporate communication costs yet. The "real" achievable speed-up can be analyzed after communication refinement on the VTA Layer.

Analyzing the Application Model, it becomes apparent that this structure is not optimal, as it does not take advantage of executing hardware and software in parallel. Therefore, we changed the initial Application Model ending up with a parallel version of the JPEG 2000 decoder ③. The processing has been parallelized between hardware and software and a pipeline structure operates in parallel on several tiles of the picture. Regrettably, this effort only has a small impact on the overall speed-up (cp. Table 8.12).

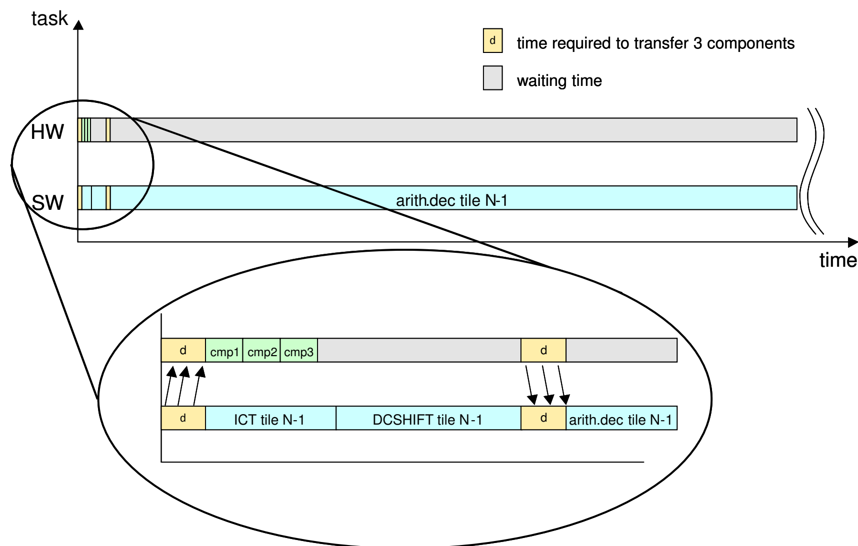


Figure 8.41: Timing diagram of JPEG 2000 decoder implementation ③ [62]

As shown in Figure 8.41 the arithmetic decoder takes up to 88% of the overall processing time, it needs to be parallelized to obtain a more significant speedup. The structure and complexity of the decoder did not allow an affordable hardware implementation. Instead we have chosen to implement it by four independent Software Tasks (cf. ④ & ⑤) performing the arithmetic decoding of disjoint parts of the image in parallel. The amount of parallel HW was not altered, since the working load of the IQ+IDWT was rather low. Figure 8.42 shows the timing diagram for the parallel and pipelined implementation ⑤.

The Application Layer models ③ & ⑤ contain two Shared Objects. The first Shared Object (IQ) handles the communication and synchronization of one ③ up to four ⑤ software tasks and three parallel hardware blocks. Moreover, it contains a data structure to transfer large objects, such as parts of the images (called tiles) and the IQ algorithm. The ability not only to store and transfer data (here tiles) but also to perform computations (here IQ) within the object was considered to be very useful. The *IDWT params* Shared Object is responsible for exchanging sequences of parameters between the control part (IDWT2D) and the lossless (IDWT53), and lossy (IDWT97) part of the IDWT. It is used not only for parameter storage and transfers, but also as arbitration unit between the three concurrent IDWT components.

After analysis, exploration and behavior refinement on the Application Layer we ended up with a design delivering an acceptable speedup by a factor of 4.5 for lossless and 5 for lossy decoding, compared to the software-only implementation. Version ⑤ suffers from the increased working load and the corresponding arbitration overhead of the hardware/software Shared

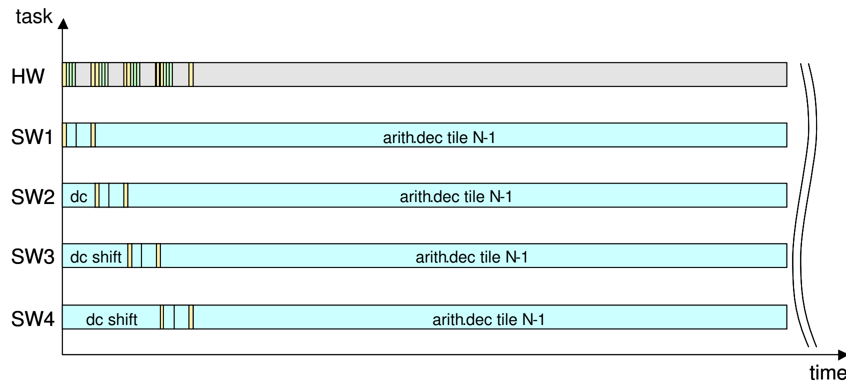


Figure 8.42: Timing diagram of JPEG 2000 decoder implementation ⑤ [62]

Object with seven clients. Hence ⑤ is slightly slower than ④.

In an intermediate version of the design example, the *IDWT params* Shared Object was not implemented. The IDWT2D was controlling IDWT53 and IDWT97 using simple RTL signals and a simple communication protocol has been developed. Even though managing communications in such a way is always possible, it is time consuming and error prone. We therefore decided for this version to use a Shared Object to manage communications between different IDWT elements.

With RTL signals, the IDWT lossless processing time is equal to 12.69ms against 14.26ms with the *IDWT params* Shared Object. The overhead is therefore of 1.57ms, which represents an increase of 11%. With RTL signals, the IDWT lossy processing time is equal to 18.02ms against 19.61ms with the *IDWT params* Shared Object. The overhead is therefore of 1.59ms, which represents an increase of 8.1%. The increase in time is therefore the same for both lossless and lossy modes, around 1.5ms, which seems to be normal. Indeed, the amount of data to be exchanged between IDWTs in lossless and lossy modes is the same and the architecture of the system is the same both in the lossless and lossy modes. Consequently, the overhead brought by the use of a Shared Object has to be the same in lossy and lossless modes.

The time required for a complete process can be split between the time required for communications (getting parameters, reading data in memory and others) and the time required by pure processing. The use of a Shared Object will increase the time required for communications. Consequently, the use of Shared Objects is well adapted in a design where communication time is non-significant compared to pure processing time.

On one hand, Shared Objects offer the possibility to connect several modules together; the designer does not have to use explicitly RTL signals and to develop any communication protocol, which results in significant reduction of development time. On the other hand, using Shared Objects slows down performances, which is partially due to arbitration and scheduling mechanisms that are embedded inside the Shared Object. On our design example, the use of a Shared Object to transfer data between IDWT modules is 1.5ms slower. Consequently, it is important to consider this overhead in order to keep the best development effort / performances ratio.

In the next step, the mapping and communication refinement process is performed to end up with a cycle-accurate Virtual Target Architecture Model.

8.7.3.2 Architecture Layer Model

In essence, the following steps have been performed to transfer the JPEG 2000 Application Model to a cycle-accurate Virtual Target Architecture Model (e.g. ③ → ⑥a).

Software Task → Software Processor: The software task is mapped onto a Software Processor of the OSSS architecture class library. This mapping is done easily by instantiating a Xilinx MicroBlaze class and mapping the software task by calling the `add_sw_task` method on the processor object.

```
1 m_microblaze_0 = new xilinx_microblaze("mb_0");
2 m_microblaze_0−>clock_port(clk);
3 m_microblaze_0−>reset_port(reset);
4 m_microblaze_0−>rmi_client_port(∗opb_bus);
5 m_microblaze_0−>add_sw_task(m_sw_task_0);
```

Following the OSSS methodology, the timing behavior can already be specified at Application Layer where it might be rather coarse. At Virtual Target Architecture Layer, it should be refined to match with the fine-grained cycle-accurate timing of the hardware and the communication model.

We have performed timing profiling on the chosen target processor running the entire JPEG 2000 decoder in software. The timing profiles have been back-annotated to the Application and the Virtual Target Architecture Models. Assuming the arithmetic decoder takes approximately 180 ms for a single tile, the following code snippet shows how to perform software timing annotations. Using the SW profiling information it makes most sense to use the so-called EET (**E**stimated **E**xecution **T**ime) blocks to annotate the execution time per function.

```
1 OSSS_EET( sc_time(180,SC_MS) ) {
2   data_send = decode_tile(Image,i+1);
3 }
```

Shared Object → Object Socket: All Shared Objects are wrapped by so called *Object Sockets*. This enables the connection to arbitrary OSSS Channels.

```
1 osss_shared<IdwtCtrl>                        IdwtCtrl_SO; //on Application Layer
2 osss_object_socket<osss_shared<IdwtCtrl> > IdwtCtrl_SO; //on VTA Layer
```

Module → Hardware Block: All modules are replaced by *Hardware Blocks*, which enable the connection to the global clock and reset signals and to arbitrary OSSS Channels.

```
1 SC_MODULE(IDWT53) {...};   //on Application Layer
2 OSSS_MODULE(IDWT53) {...}; //on VTA Layer
```

Data serialization: The serialization cuts large user-defined data structures into manageable chunks of data to be transferred efficiently via OSSS Channels. The OSSS modeling library provides pre-defined methods, which can be re-used via inheritance to implement the serialization method.

```
1 void serialise () {
2   osss_serialisable::store_element(m_tile);
3   osss_serialisable::store_array(m_data, 16);
4 }
```

Explicit memory insertion: Some data members - especially large arrays - in HW/SW Shared Objects should be mapped into explicit memory (in the following listing: Xilinx Block RAM with 32 bit data and 16 bit address width). In the VTA Model it is important to assess the effects of data locality in order to reach the best area/performance trade-off for the implementation. If data were not stored explicitly into such memories, it would be synthesized as fast registers and dramatically increase the amount of occupied FPGA slices.

```
1 osss_array<short,2∗N+5>                            m_array; //on Application Layer
2 xilinx_blockram<osss_array<short,2∗N+5>, 32, 16> m_array; //on VTA Layer
```

Communication Link → RMI Channel: In this refinement step multiple communication links can either be bundled in a physical shared bus or each communication link can be mapped to a dedicated point-to-point connection.

In model ⑥a/⑦a all communication links to the hardware/software Shared Object (IQ) are mapped to an OSSS Channel implementing an IBM OPB. Communication links to the *IDWT Params* Shared Object are mapped to dedicated point-to-point channels.

All method calls that have been performed through communication links in the Application Layer Model are now performed through OPB or point-to-point channels using the OSSS RMI protocol.

The RMI Channels, implementing the RMI protocol on top of OSSS Channels, provides a simple mechanism to map the method-based communication onto different physical communication channels. This enables the exploration of alternative mappings leading to more efficient solutions. For example, model (6b)/(7b) uses an alternative mapping where only point-to-point channels are used to implement the communication of the IDWT hardware blocks with the Shared Object.

The lower part of Table 8.12 shows the impact of the VTA refinements on the simulation results. Now we compare the pure Application Layer Models with the corresponding VTA Layer mappings:

- ③ → (6a)/(6b): The IDWT time is increased significantly (up to a factor of 8). This increase in time results directly from the channel refinement and the explicit memory insertion in the VTA model. Anyway, this version is dominated by the software part and therefore the overall decoding time is not affected significantly.
- ⑤ → (7a)/(7b): In (7a) the IDWT time is increased even more than in (6a) since three more processors are competing for access to the single shared bus. The IDWT times of (6b) and (7b) are equal since in both VTA models the same P2P connections are used and the hardware/software Shared Object (IQ) decouples the bus accesses initiated by the software tasks.

Having a look at Table 8.12 we observe the impact of the VTA refinements on the simulation results. Compared to the simulation models on Application Layer the overall decoding time is increased by 2.8% in the worst and 0.05% in the best case. The increase in the fraction of the IDWT time results directly from the channel refinement and the explicit memory insertion in the VTA model. Figure 8.43 shows a comparison of the AL to VTAL overhead between model ③ and models (6a)/(6b).

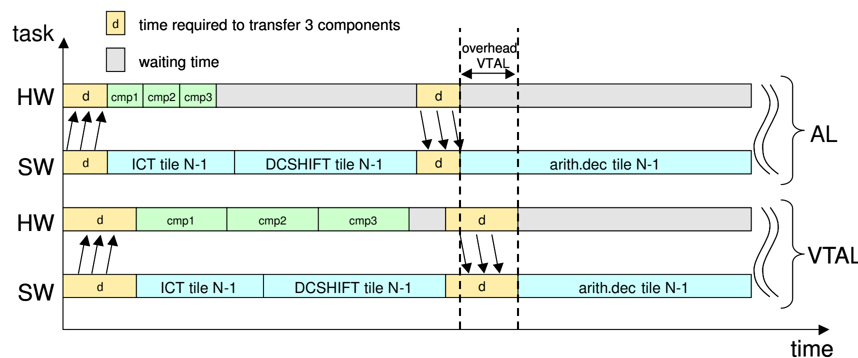


Figure 8.43: Timing diagram of JPEG 2000 decoder comparing the AL to VTAL overhead between ③ and (6a)/(6b) [62]

With all refinements we still observe a speed-up by a factor of 12 (lossless) and 16 (lossy) for the IDWT in hardware (6b)/(7b) compared to the software only execution in ①. The exploration on the VTA Layer has confirmed our assumptions obtained from the Application Layer experiments. It shows that the overall processing time is dominated by the SW arithmetic decoder. Even after software parallelization, the working load of the shared hardware IDWT does not exceeded its limit, even when using a single shared bus. This points out that (7a) is an affordable implementation of the JPEG 2000 decoder while (7b) does better scale with increasing parallelism.

8.7.4 Implementation Models

Figure 8.44 shows the OSSS synthesis flow for the JPEG 2000 decoder design. For synthesis, we incorporate the behavior of the system as defined by the Application Layer Model, the allocated HW resources of the VTA Layer and the mapping information between them.

During the synthesis process, the VTA Layer is analyzed and separated into a software and a hardware subsystem. The software subsystem in our design consists of the Xilinx MicroBlaze

Version of JPEG Decoder Model	Decoding Time[a] [ms] lossless (speedup)	Decoding Time[a] [ms] lossy (speedup)	IDWT Time[a] [ms] lossless	IDWT Time[a] [ms] lossy
Application Layer				
① SW only on MicroBlaze	3188.79 (1)	2662.76 (1)	173.11	323.13
② HW/SW not parallel	2847.03 (1.12)	2144.96 (1.24)	1.33	6.62
③ HW/SW parallel (3 IDWT modules)	2845.08 (1.12)	2138.21 (1.25)	3.77	9.12
④ SW parallel (cp. ②)	712.11 (4.48)	537.62 (4.95)	1.33	6.62
⑤ SW & HW/SW parallel (cp. ③)	721.08 (4.42)	550.22 (4.84)	3.77	9.12
Virtual Target Architecture Layer				
HW/SW parallel				
(6a) HW/SW SO connected to bus only	2848.22 (1.12)	2141.38 (1.24)	30.94	36.31
(6b) HW/SW SO connected to bus & P2P	2846.48 (1.12)	2139.62 (1.24)	14.26	19.61
SW & HW/SW parallel				
(7a) HW/SW SO connected to bus only	732.11 (4.35)	558.29 (4.77)	55.99	61.27
(7b) HW/SW SO connected to bus & P2P	722.21 (4.42)	551.34 (4.83)	14.26	19.61

[a] time needed to decode 16 tiles with 3 components each @ 100 MHz

Table 8.12: Simulation results [45]

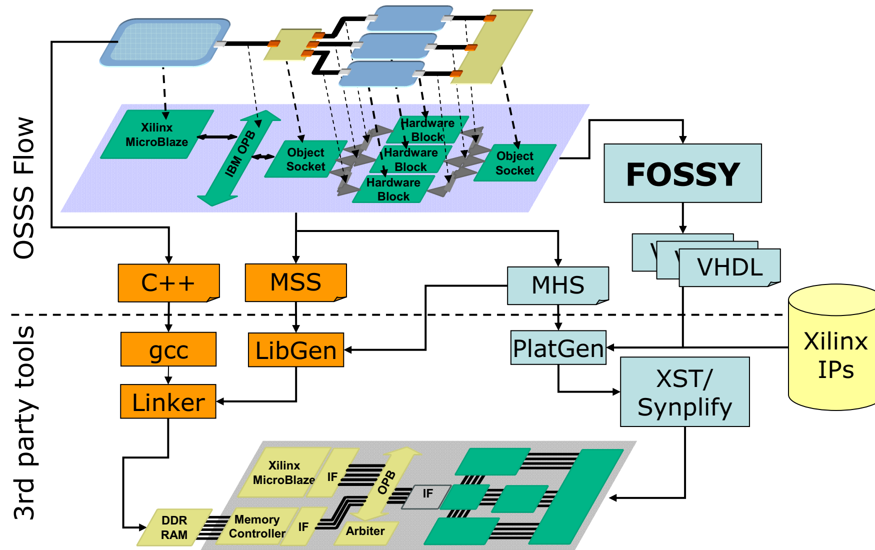


Figure 8.44: Synthesis flow [45]

together with its associated software task and the OPB. The hardware subsystem is made up of two Object Sockets containing the hardware/software and the hardware/hardware Shared Object, three IDWT modules and OSSS Channels for communication between them.

Since our prototypical synthesis flow interfaces the Xilinx Embedded Development Kit (EDK) we are generating vendor-specific architecture definition files. These are the MSS (**M**icroprocessor **S**oftware **S**pecification) and MHS (**M**icroprocessor **H**ardware **S**pecification) file that are used for the creation of an EDK project.

For the implementation of the JPEG 2000 case-study we are using the MicroBlaze, the OPB and an OPB multi-channel DDR-RAM controller from the Xilinx EDK IP core library.

The software tasks are cross-compiled and linked against a specific OSSS embedded library that enables the communication with the hardware/software Shared Object. The entire hardware subsystem is transformed from SystemC/OSSS to synthesizable VHDL code by *Fossy*. The resulting VHDL code is inserted into the generated EDK project and further processed by the Xilinx Synthesis Tool (XST) or other third-party RTL synthesis tools.

In the following we are going to have a closer look on the *Fossy* generated VHDL code of

	lossless (IDWT53)[b]		lossy (IDWT97)[b]	
	Fossy	reference	*Fossy*	reference
Logic Utilization				
Number of Slice Flip Flops	298	293	453	674
Number of 4 input LUTs	682	619	1741	1848
Logic Distribution				
Number of occupied Slices	436	356	944	1077
Total Number 4 input LUTs	730	634	1747	1878
Number used as logic	682	619	1741	1848
Number used as a route-thru	48	15	6	30
Total equivalent gate count	7882	7146	16052	18885
Estimated max. frequency (f_{max}) [MHz]	212.1	212.4	136.7	189.3

[b]with Xilinx ISE 9.2.02i for Virtex-4 LX25 FPGA

Table 8.13: RTL Synthesis results of the IDWT [45]

the IDWT modules.

8.7.4.1 IDWT Reference Models

A wavelet transform improves encoding efficiency by exploiting pixel correlation. It is composed of a set of filters that extract parts of the original signal: low-pass filters preserve a blurred representation of the original signal, while high-pass filters preserve transitions and textures of the image. On the decoder side, the Inverse Discrete Wavelet Transform reconstructs the image as shown in Figure 8.39. In the lossless mode (IDWT53) the used filter-bank is a bi-orthogonal (5,3) with integer taps, and in the lossy mode (IDWT97) it is a Daubechies (9,7).

The handcrafted reference model is written in synthesizable VHDL and consists out of 404 (IDWT53) and 948 (IDWT97) lines of code. Table 8.13 shows the result of the RTL synthesis performed by the Xilinx Synthesis Tool (XST) of ISE 9.2.02i for a Virtex-4 LX25 FPGA.

8.7.4.2 *Fossy* Generated Models

The SystemC IDWT models used for simulation and exploration in Section 8.7.3 are described at the same level of abstraction as the reference model, i.e. RTL. This is necessary at VTA Layer at least to get a more realistic estimate of the system behavior after implementation.

The synthesizable SystemC IDWT models consist out of 356 (IDWT53) and 903 (IDWT97) lines of code. The overall structure of the SystemC and the reference VHDL model is very similar. Both use explicit state machines and functions and procedures to separate the more complex filter algorithms from the control dominated part. The *Fossy* generated VHDL models consist out of 2231 (IDWT53) and 4225 (IDWT97) lines of code where all functions and procedures have been inlined into a single explicit state machine. Since all identifiers are preserved during synthesis the resulting VHDL code remains human readable.

8.7.4.3 Comparison

Table 8.13 shows the RTL synthesis results for a Xilinx Virtex-4 FPGA. For the IDWT53 the results are very similar. The area overhead induced by *Fossy* is about 10%. Concerning the IDWT97 the *Fossy* generated design is 15% smaller but 28% slower than the reference. Since the JPEG 2000 decoder is a hardware/software design where both the MicroBlaze and the OPB run with a frequency of 100 MHz the synthesis results perfectly match the timing requirements.

The performance of the IDWT97 *Fossy* generated design can be improved by adding certain register levels. Furthermore, the ISE was not capable of synthesizing pre-adders of the desired input bit widths.

8.7.5 Conclusion

In this experiment we have presented the modeling and design of a JPEG 2000 decoder using the OSSS design methodology. The initial Application Model specifies the logical structure of the application, which identifies parallel components and their communication relations. The Application Model enables early functional and performance analysis in a rather abstract thus fast simulation model. Moreover, the designer can easily restructure the model, e.g. the hardware/software partitioning and assess the effect on the behavior and estimated performance of the design.

The refinement process leading to the Virtual Target Architecture Model adds implementation details such as communication protocols and memories. This model exhibits cycle-accurate timing behavior and is the input for the automatic synthesis process leading to the implementation on a specific target platform.

The comparison of the synthesis results using the *Fossy* synthesis tool or a standard VHDL design process shows no significant difference in the efficiency in terms of area or timing.

8.8 Summary

In this chapter, the OSSS methodology has been applied to a set of different designs. An overview of these designs can be found in Table 8.1. With the experiments the majority of goals from Chapter 2 have been met. To summarize the experiments, Table 8.14 presents a review and discussion of covered goals.

Table 8.14: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled)

ID	Goal	Status	Comment
G2	Introduce a notion of time for the SW parts	●	Estimated Execution Time (EET) blocks, as described in Section 6.4.3.3, enable timing annotation of Software Tasks. Required Execution Time (RET), as described in Section 6.4.3.4, enable dynamic run-time checks of software timing requirements.
M1	Single modeling language to describe HW and SW	●	OSSS, as introduced in Section 6.2 covers the description of HW and SW. On the Application Layer, SW is described by Software Tasks (see Section 6.4.3) and HW is described by Hardware Modules (see Section 6.4.5).
M3	Executable Specification and HW/SW partitioned models	●	The OSSS simulation model covers the untimed specification level modeling (OSSS Behavioural Layer, see Section 6.3), timed HW/SW partitioned modeling (OSSS Application Layer, see Section 6.4) and timed execution platform modeling (OSSS Virtual Target Architecture Layer, see Section 6.5).
M4	Synthesizable HW/SW partitioned model	●	An OSSS Application Layer model (which describes a HW/SW partitioned model) mapped to an OSSS Virtual Target Architecture Layer model is synthesizable with the prototypical synthesis tool *Fossy*, see Chapter 7.

continued on next page

Table 8.14: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
M5	To be able to cover untimed (purely functional) models, transaction-level models and cycle accurate models	●	The Behavioural Layer (see Section 5.4) enables untimed (purely functional) system modeling. The Application Layer (see Section 5.5) enables transaction-level modeling, because communication between Actors and Shared Objects is performed by abstract service calls. Application Layer models mapped to Virtual Target Architecture Layer models (see Section 5.6) enables cycle accurate system modeling.
M7	Easy HW/SW repartitioning of the design (a SW module can be replaced by a HW module without manually modifying its communication interfaces)	●	At the Application Layer, Actors are used to model HW and SW components. The have the same connections (using the same port to interface binding concept) to Shared Objects. When replacing an Actor modeling a Software Task with an Actor modeling a Hardware Module no modification on the other components (incl. Shared Objects) of the Application Layer become necessary.
M10	Possibility to write hardware modules at RT-level	●	HW Modules at the Application Layer (see Section 6.4.5) are described at behavioral RT (using `SC_CTHREAD`s). For more details see Chapter F.
M14	Integration of IP components	●	IP components can be integrated using IP Component wrapper modules. Signal based communication with Hardware Modules and Shared Objects plugged into Adapter Sockets is supported.
A2	High simulation performance (at least higher that state-of-the-art RTL simulations)	◖	Our experiment in Section 8.2 show that the simulation performance of our simulation model is better than the performance obtained with the SpecC reference simulator. In our experiment in Section 8.4 we have compared the simulation speed of a design on the OSSS Application Layer and OSSS Virtual Target Architecture Layer with the simulation speed of the synthesized VHDL and a reference VHDL design simulated using ModelSim 6.1e. The Application Layer model simulation has been more than 5 times and the Virtual Target Architecture Layer model still more than 3 times faster than the reference VHDL model simulation in ModelSim 6.1e.
A3	Basic timing properties shall be reflected by the simulation	●	On the Application Layer timing properties of Software Tasks are represented by EET blocks and timing properties of Hardware Modules are represented by SystemC `wait()` statements. On the Virtual Target Architecture Layer timing properties of the RMI protocol and the on-chip communication resources (bus, point-to-point) are added.

continued on next page

Table 8.14: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
A4	Combine models of different levels of abstraction in a single simulation	◖	In OSSS Behavioural Layer and Application Layer modeling elements can be simulated together. The enables stepwise refinement from an untimed to a timed simulation model. Application and Virtual Target Architecture Layer modeling elements cannot be mixed (i.e. all Application Layer elements need to be mapped to their corresponding Virtual Target Architecture elements). On the Application Layer it is possible to combine timing approximate models with cycle-accurate models.
S6	For the integration of IP components it is necessary that the designer can control the synthesis and to enforce a certain communication mechanism, which is required by the IP component	●	The supported modeling style of Hardware modules is behavioral RTL which enables a clock-cycle accurate protocol description suitable for RTL IP component integration.
S8	Efficiency of the generated code (for hardware: area and critical path; for software: memory footprint) compared to a hand-crafted design	◖	The efficiency of the generated custom hardware VHDL code in terms of area and critical path length, mainly depends on the SystemC input code. We have shown for an industrial use-case (see Section 8.4) that the *Fossy* generated VHDL code has an area overhead of 16% and a maximum clock frequency reduced by 3% compared to a hand optimized VHDL design.

CHAPTER 9

Conclusion

The OSSS methodology, as presented in this work, supports many useful object-oriented design concepts such as Actors, Objects, Shared Objects, method-based communication. By adding a representation of the target hardware platform structural representation, Actors, Objects and Shared Objects are mapped to specific hardware resources and with the aid of a synthesis tool, a direct path to the implementation on a FPGA has been presented.

Figure 9.1 gives an overview of the core OSSS modeling layers and the Shared Object and interface synthesis.

The OSSS Behavioral Layer (not shown in Figure 9.1) and the OSSS Application Layer are the design entries of OSSS. The Behavioral Layer enables design capturing using an object-oriented Program State Machine (PSM) model of computation, inspired by SpecC. For synthesis, a Behavioral Layer model needs to be refined to an Application Layer model first. For this refinement step, pre-defined Shared Object templates are provided.

The Application Layer enables modeling of a refined functional specification that enables the differentiation between hardware and software and its communication. "Active" modeling elements are Actors, which can be Software Tasks or Hardware Modules, and "Passive" modeling elements are user-defined Objects and Shared Objects. Active modeling elements can only communicate by using service calls on Shared Object. The service calls are user-defined transactions that enable to transport any user-defined Object or data type from an Active modeling element to a Shared Object and vice versa. Thus, the Application Layer only defines a logical communication architecture with logical communication links between Actors and Shared Objects.

For implementing an OSSS Application Layer design on an SoC platform, it needs to be mapped and refined to the Virtual Target Architecture Layer (VTA). The VTA represents the structural hardware resource description of the target SoC. In OSSS this structural resource representation abstracts from many hardware platform details using a so-called Architecture Class Library with generic hardware resource representatives. Theses are Hardware and Software Sockets and Remote Method Invocation (RMI) Channels. Hardware Sockets are representing custom hardware blocks with RMI interface (implemented in custom hardware), like an ASIC of resources of an FPGA. Software Sockets are representing a processor and its run-time system supporting the RMI protocol. The RMI Channel represents the structural on-chip communication network, such as a shared bus or dedicated point-to-point connection (expressed as inner channel using OSSS Channel description). During refinement, components of the Application Layer are mapped to the VTA. Software Tasks are mapped onto Software Sockets, Hardware Modules are mapped onto Hardware Sockets, and Shared Objects are mapped onto Shared Object Sockets, which as special Hardware Sockets. The logical or abstract communication links between Actors and Shared Objects in the Application Layer model are implemented using the RMI protocol over a specific interconnect infrastructure, like buses, point-to-point channels as defined in the

VTA. Thus, method calls from Actors to Shared Objects are automatically refined to an RMI Message Passing protocol implementation. Different VTA configurations and Application Layer mappings enable platform exploration.

To finally implement the Application Layer model to VTA Layer mapping on the target platform, automatic synthesis support is provided. For this purpose, the FOSSY (Functional Oldenburg System Synthesiser) high-level synthesis tool transforms the Shared Object representations of the VTA into synthesizable VHDL models. In this synthesis process, the Shared Object is split-up into multiple modules responsible for arbitration, scheduling and the behavior. The RMI message-passing interface is replaced by a signal-based interface. This signal interface allows integration into existing SoC through provision of a generic, bus-independent interface (cp. Xilinx IPIF). Furthermore, it allows user-defined communication channel bit-width constraints on point-to-point channels for supporting constraints on routing resources between different design partitions, a constraint number of I/O pins (inter SoC communication), a specific area vs. throughput constraint. The provided synthesis infrastructure enables the integration of vendor specific back-end synthesis tools and flow. In this work only the Xilinx FPGA back-end synthesis tool-chain has been presented. These are the Xilinx EDK/ISE tools for platform generation, low-level synthesis, mapping, and place & route, cross compilation of the software part of the design, and finally bit-stream initialization and download to the FPGA.

The effectiveness of the presented OSSS methodology and tool-chain has been evaluated by and shown a set of different design examples.

Section 9.1 summarizes the review of all goals from Chapter 2 and Section 9.2 discusses the limitation and possible future work.

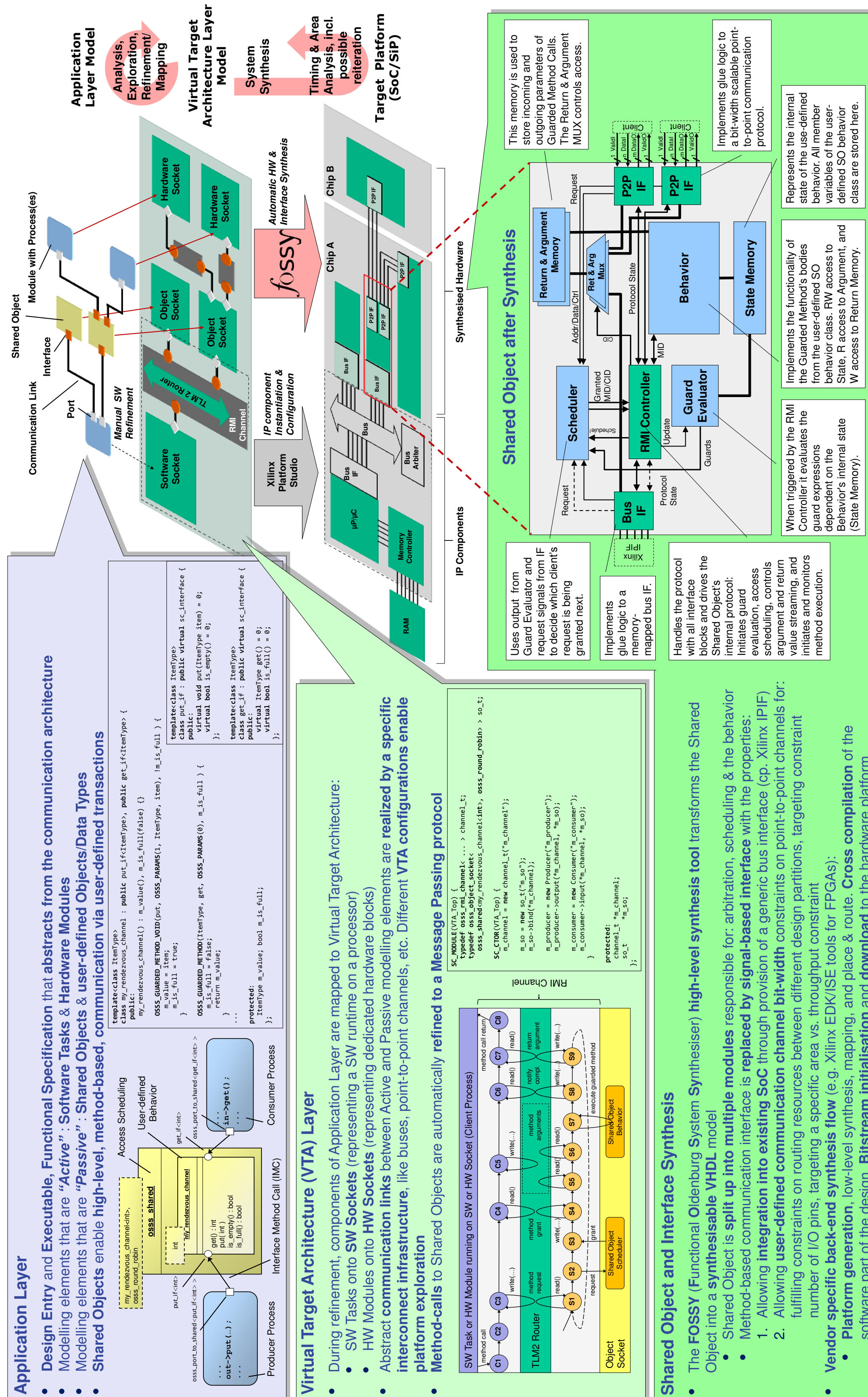


Figure 9.1: Overview of the core OSSS modeling layers, Shared Object and Interface Synthesis

9.1 Review of Goals

In Table 9.1 this section summarizes the review of all goals from Chapter 2. In Section 9.2 the limitations of the current work, as also shown in Table 9.1, are discussed and possible future work is described.

Table 9.1: Review of all goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled)

ID	Goal	Status	Comment
G1	Integration of synthesis tool and simulation infrastructure into Eclipse CDT Framework	●	Since OSSS is based on C++ and integration into the Eclipse C/C++ Development Tooling (CDT) Framework is possible. Both, the simulator and the synthesis tool *Fossy* have been successfully integrated, see Chapter G.
G2	Introduce a notion of time for the SW parts	●	Estimated Execution Time (EET) blocks, as described in Section 6.4.3.3, enable timing annotation of Software Tasks. Required Execution Time (RET), as described in Section 6.4.3.4, enable dynamic run-time checks of software timing requirements.
M1	Single modeling language to describe HW and SW	●	OSSS, as introduced in Section 6.2 covers the description of HW and SW. On the Application Layer, SW is described by Software Tasks (see Section 6.4.3) and HW is described by Hardware Modules (see Section 6.4.5).
M2	SystemC approach	●	OSSS is based on SystemC as described in Section 6.2.
M3	Executable Specification and HW/SW partitioned models	●	The OSSS simulation model covers the untimed specification level modeling (OSSS Behavioural Layer, see Section 6.3), timed HW/SW partitioned modeling (OSSS Application Layer, see Section 6.4) and timed execution platform modeling (OSSS Virtual Target Architecture Layer, see Section 6.5).
M4	Synthesizable HW/SW partitioned model	●	An OSSS Application Layer model (which describes a HW/SW partitioned model) mapped to an OSSS Virtual Target Architecture Layer model is synthesizable with the prototypical synthesis tool *Fossy*, see Chapter 7.
M5	To be able to cover untimed (purely functional) models, transaction-level models and cycle accurate models	●	The Behavioural Layer (see Section 5.4) enables untimed (purely functional) system modeling. The Application Layer (see Section 5.5) enables transaction-level modeling, because communication between Actors and Shared Objects is performed by abstract service calls. Application Layer models mapped to Virtual Target Architecture Layer models (see Section 5.6) enables cycle accurate system modeling.

continued on next page

Table 9.1: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
M6	Methodology needs to provide modeling elements which allow to describe the communication	●	At the Behavioural Layer pre-defined channels for the communication between Behaviors are provided (see Section 5.4.2.3). At the Application Layer, Shared Objects are provided for the communication between Actors (see Section 5.5.2.3). At the Virtual Target Architecture Layer OSSS-RMI Channel (see Section 5.6.2.8) containers and OSSS-Channels (see Section 5.6.2.9) are provided for modeling communication at the SoC platform.
M7	Easy HW/SW repartitioning of the design (a SW module can be replaced by a HW module without manually modifying its communication interfaces)	●	At the Application Layer, Actors are used to model HW and SW components. The have the same connections (using the same port to interface binding concept) to Shared Objects. When replacing an Actor modeling a Software Task with an Actor modeling a Hardware Module no modification on the other components (incl. Shared Objects) of the Application Layer become necessary.
M8	Provide constructs for a uniform interface description	●	At the Behavioural Layer, channels with pre-defined interface services are provided. At the Application Layer, Shared Objects with user-defined service interfaces are provided. During mapping of the Application to the Virtual Target Architecture the Shared Object service interfaces are preserved.
M9	Provide modeling constructs for abstract communication	●	At the Behavioural Layer pre-defined channels for the communication between Behaviors are provided (see Section 5.4.2.3). At the Application Layer, Shared Objects are provided for the communication between Actors (see Section 5.5.2.3). Both communication concepts abstract from the signal level implementation of the communication. At the Virtual Target Architecture Layer service calls on Shared Objects are implemented by RMI-Channels and OSSS-Channels.
M10	Possibility to write hardware modules at RT-level	●	HW Modules at the Application Layer (see Section 6.4.5) are described at behavioral RT (using `SC_CTHREAD`s). For more details, see Chapter F.
M11	Support of multitasking	○	Not supported in this work. Extensions of OSSS for software multitasking can be found in [48, 23, 17].
M12	Consideration of (real-)time constraints	◖	The combination of EETs and RETs can be used to specify real-time constraints. Currently RETs can only be used within one Actor. This modeling restriction restricts the expressiveness of advanced (real-)time constraints, e.g. end-to-end deadlines.
M13	Support of operating systems	○	Not supported in this thesis (see M11).

continued on next page

Table 9.1: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖ : partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
M14	Integration of IP components	●	IP components can be integrated using IP Component wrapper modules. Signal based communication with Hardware Modules and Shared Objects plugged into Adapter Sockets is supported.
A1	Debugging on all levels of abstraction	◖	Debugging of OSSS simulation models is not directly supported by the methodology or simulation library. Nevertheless, since OSSS builds on top of SystemC and C++, common waveform visualization and debugging tools can be used.
A2	High simulation performance (at least higher that state-of-the-art RTL simulations)	◖	Our experiment in Section 8.2 show that the simulation performance of our simulation model is better than the performance obtained with the SpecC reference simulator. In our experiment in Section 8.4 we have compared the simulation speed of a design on the OSSS Application Layer and OSSS Virtual Target Architecture Layer with the simulation speed of the synthesized VHDL and a reference VHDL design simulated using ModelSim 6.1e. The Application Layer model simulation has been more than 5 times and the Virtual Target Architecture Layer model still more than 3 times faster than the reference VHDL model simulation in ModelSim 6.1e.
A3	Basic timing properties shall be reflected by the simulation	●	On the Application Layer timing properties of Software Tasks are represented by EET blocks and timing properties of Hardware Modules are represented by SystemC `wait()` statements. On the Virtual Target Architecture Layer timing properties of the RMI protocol and the on-chip communication resources (bus, point-to-point) are added.
A4	Combine models of different levels of abstraction in a single simulation	◖	In OSSS Behavioural Layer and Application Layer modeling elements can be simulated together. The enables stepwise refinement from an untimed to a timed simulation model. Application and Virtual Target Architecture Layer modeling elements cannot be mixed (i.e. all Application Layer elements need to be mapped to their corresponding Virtual Target Architecture elements). On the Application Layer it is possible to combine timing approximate models with cycle-accurate models.
A5	Consideration of IP components in the simulation	◖	SystemC RTL IP components can be integrated on the Application Layer. VDHL or Verilog RTL IP components can only be integrated when using a SystemC and VHDL or Verilog co-simulation environment (e.g. ModelSim).

continued on next page

Table 9.1: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
S1	Provide a (prototypical) synthesis tool	●	*Fossy* is a prototypical synthesis tool Shared Object and OSSS Hardware Module to VHDL synthesis, configuration and integration of these custom hardware blocks into an SoC architecture for Xilinx FPGAs.
S2	Software output language C++ compliant with C++ standard (ISO/IEC 14882:1998)	◖	In the current implementation *Fossy* does not generate any software code. The designer needs to write C++ code inside the Software Modules which is compliant with ISO/IEC 14882:1998.
S3	Hardware language VHDL compliant with the synthesizable subsets of Synopsys Design Compiler and Synplify Pro from Synplicity	◖	The *Fossy* generated VHDL code has been successfully synthesized with Synplify Pro a Xilinx FPGA. The Synopsys Design Compiler for an ASIC target has not been tested so far.
S4	The generated code has to be readable for a human being	◖	The *Fossy* generated VHDL and SystemC code is human readable. All identifier names are preserved (some of them with pre-fixes). Type transformation (which is necessary to map the SystemC data type semantics to the VHDL semantics) introduces some additional casts. The state-machine transformation which translates implicit to explicit state-machines transforms `SC_CTHREADS` into `SC_MODULE` introducing explicit states states and next-state logic. With these transformations the code should stay human readable.
S5	Possibility to map the abstract communication objects onto concrete mechanisms such as memory mapped IO/ shared memory (using polling, interrupts and/or DMA) or proprietary direct HW/HW communication and to generate the necessary HW and SW parts	◖	OSSS Application Layer Communication Links can be mapped to a bus using memory mapped IO (currently only polling access is supported) or to any proprietary direct HW/HW communication. Interrupts and DMA are currently not supported.
S6	For the integration of IP components it is necessary that the designer can control the synthesis and to enforce a certain communication mechanism, which is required by the IP component	●	The supported modeling style of Hardware modules is behavioral RTL which enables a clock-cycle accurate protocol description suitable for RTL IP component integration.

continued on next page

Table 9.1: Review of selected goals from Chapter 2 (G: general, M: modeling, A: analysis, S: synthesis, ●: fulfilled, ◖: partly fulfilled, ○: not fulfilled) – *continued*

ID	Goal	Status	Comment
S7	Control of the synthesis by constraints in the synthesis script or by special statements within the source code	○	Currently not supported by *Fossy*.
S8	Efficiency of the generated code (for hardware: area and critical path; for software: memory footprint) compared to a hand-crafted design	◖	The efficiency of the generated custom hardware VHDL code in terms of area and critical path length, mainly depends on the SystemC input code. We have shown for an industrial use-case (see Section 8.4) that the *Fossy* generated VHDL code has an area overhead of 16% and a maximum clock frequency reduced by 3% compared to a hand optimized VHDL design.

9.2 Limitations and Future Work

As already identified in Table 9.1, the presented OSSS methodology has the following limitations. Some of them have already been addressed in current work, others are still open for future work.

Non-blocking and streaming method calls In this work Shared Object method calls are always blocking. A client process is actively waiting until the method call is completed and returns. This limits the amount of possible parallelism and the maximum possible utilization utilization of the computational resource, the client process is running onto.

Non-blocking method calls would allow the client process to continue its computation after the method call has been sent to the Shared Object. The use of *Futures* as method return container type, enables to access the result of the method call, when it is actually required. In this case, the immediate access to the Future container object, after the method call has been issued, would be the same like a blocking method call. This way, blocking method calls would become a special case of the non-blocking method call mechanism. When using non-blocking method calls with Future container objects a decision, whether to allow nested method calls must be taken, since this might lead to Future object resource allocation problems, when the depth of the nested calls is not bounded and cannot be statically obtained during design time.

Streaming method calls could also be called "fire and forget" method calls. A client calls a streaming method on a Shared Object and does not wait for the method to complete. The access to the Shared Object is blocked until the arguments of the previous streaming call have been processed. For an efficient implementation of streaming method calls, the behavior of the Shared Object and the RMI protocol machine should be able to work in parallel (i.e. as a pipeline) as well.

Both, the *non-blocking* and the *streaming* method call mechanisms have not been implemented yet and are subject to future work.

Call by reference In this work Shared Object method calls only allow to pass method arguments via call by value. In a call by value method call, the arguments are copied from the client process to the Shared Object and the return value is copied back to the client process. Using *call by reference* semantics would enable to use references to data object containers inside a memory of the target platform. This way the client process would pass only the information of the arguments memory location to the Shared Object. With this information, the Shared Object would be able to fetch the data from this memory location and store back the results in the same memory location (only if the reference is non-const). This mechanism would relieve the client process from actively copying method call arguments

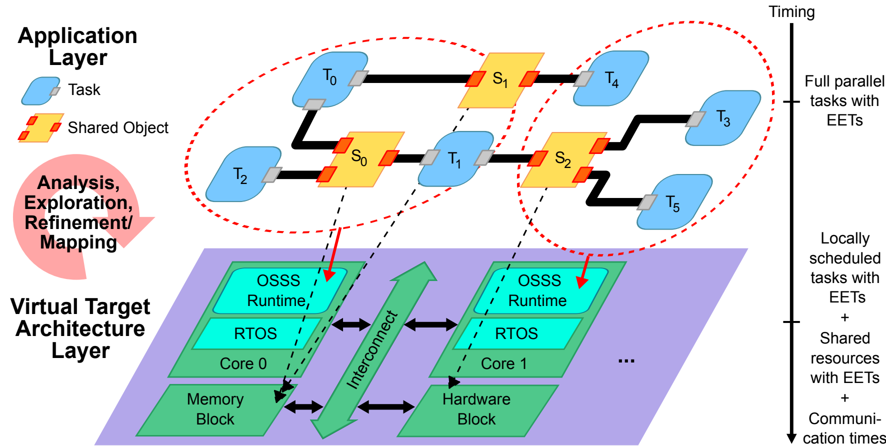


Figure 9.2: Extension of OSSS for modeling parallel software (Source: [23])

to the Shared Object. Especially in the case of a client, running on a software processor this can be very inefficient, since data that is already present at a shared memory location would be unnecessarily copied.

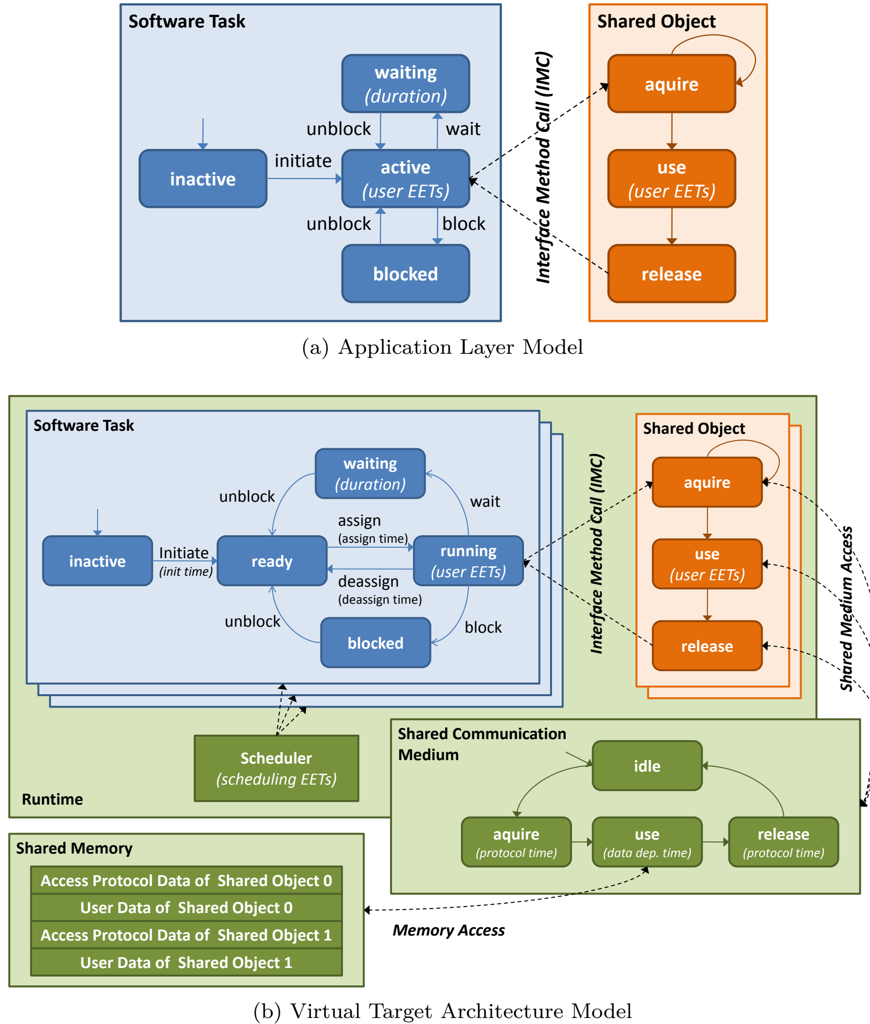
The drawback of the call by reference semantics is the increased complexity of the Shared Object, since it has to be able to actively fetch data from various memory locations, which also requires the Shared Object to implement a bus master interface with burst capabilities. Furthermore, a design time check needs to guarantee that the argument container objects memory locations can be logically and physically reachable by the Shared Object. Another challenge is concerning the memory layout consistency. As processors usually use a data layout that enables fast access by their supported ISA. This way, either shared data must be represented in a normalized layout, accessible by hardware and software, or the Shared Object needs to know the memory layout rules of the used software processor. In a platform with heterogeneous software processors with different memory layouts this might be challenging. However, the normalized layout might imply a too inefficient access from the software. Nevertheless, this depends on the actual usage of the shared memory.

Due to the described challenges the *call by reference* semantics has not been considered in this work and is subject to future work.

Software multi-tasking has not been addressed in this work. In [48] an extension of this work regarding software multi-tasking has been published for the first time. It introduces a software run-time model supporting different scheduling policies, as well as efficient timing annotations, and deadlines. SW/SW Inter-task communication is modeled via Shared Objects. The modeling primitives like Software Tasks and Shared Objects are similar to the elements on the OSSS Application Layer (used in this work) and abstract from explicit and error-prone synchronization primitives, the underlying RTOSs would provide. The integrated RTOS abstraction includes different scheduling policies (preemptive and cooperative), periodic and continuous tasks, priorities, absolute and relative deadlines, without being targeted to a specific RTOS directly. As long as some locking primitive is available on the software target architecture, the OSSS software run-time can be mapped on this platform. A prototypical implementation on an existing RTOS has been published in [17]. The HW/SW and SW/HW communication capabilities of OSSS have not been fully integrated with the software multi-tasking implementation in [48]. The communication refinement still follows the RMI and OSSS Channel approach as presented in this work.

In [23] an extension of [48] to support multi-core architectures has been published. Here, the OSSS model is *not* meant to directly represent existing real-time operating system (RTOS) primitives. Instead, the *Software Tasks* in OSSS are meant to *run* on top of a rather generic (but lightweight) run-time system (see Figure 9.2), where the synchronization and inter-task communication is modeled with *Shared Objects*.

In a refinement step the *Application Layer* model is mapped to the *Virtual Target Architecture*. Each task is then mapped to a specific core, each of which provides a distinct run-time, as shown in Figure 9.2. Tasks have may have statically or dynamically assigned priorities, according to a given scheduling policy for each core, an initial startup time, optional periods and deadlines.



(a) Application Layer Model

(b) Virtual Target Architecture Model

Figure 9.3: Task states and transitions (terminate edges omitted) (Source: [23])

During simulation, the tasks can be in different states as shown in Figure 9.3. We distinguish between the full parallel *Application Model* and the core mapped *Virtual Target Architecture Model* task state machines. In the *Application Model* a task can either be `running`, `waiting` or `blocked`. The distinction between `blocked` and `waiting` has been introduced to ease the detection of deadlocks. A task in the `waiting` state will enter the `running` state after a given amount of time (duration), whereas a `blocked` task can only be de-blocked, once the access to a shared resource is granted. In the `running` state, a task might access a *Shared Object* through Interface Method Call (IMC). This either leads to the acquisition of its critical section (`use` state) or a suspension in the `blocked` state. In this state the task tries to reacquire the shared resource until it gets access.

In the *Virtual Target Architecture Model Software Tasks* and *Shared Objects* are grouped and mapped onto run-times of the cores. During the simulation, the OSSS software run-time abstraction handles the time-sharing of a single processor core by several *Software Tasks*, which are bound to this OS instance. Therefore, a `ready` state has been introduced. A scheduler for handling the time-sharing is attached to the set of mapped tasks. Several

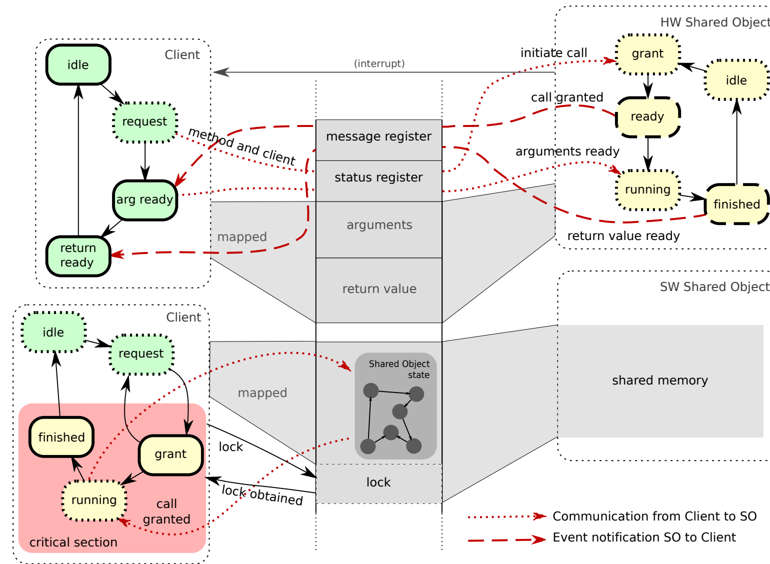


Figure 9.4: Extended interaction and memory layout of Client and Shared Object (Source: [17])

frequently used scheduling policies are already provided by the extended simulation library, like static priorities (preemptive and cooperative), or earliest-deadline first. Additionally, arbitrary user-defined scheduling policies can be added. The RTOS overhead of context switches (assign & deassign times) and execution times of scheduling decisions can be annotated as well. With this set of basic elements, the behavior of the real RTOS on the target platform can be modeled.

To improve the real-time capabilities, *Guarded Methods* that can lead to arbitrary blocking times due to data-dependent conditions, are ignored. Instead, only the guaranteed mutual exclusive access to Shared Objects is used for synchronization and communication between the tasks. Each method of such a Shared Object can then be considered as a critical section, which is executed atomically. Intra-core communication, i.e. communication between tasks mapped to the same core, can be handled as usual. Here, the accesses are ordered according to the local scheduling policy.

Moreover, the *Virtual Target Architecture Model* allows incorporating the effects of a shared memory that is connected to the cores via a shared communication medium. In an implementation on a target architecture the access protocol data, as well as the user data of a *Shared Object* are mapped to a specific location in a shared memory. Therefore, all states of the *Shared Object* include a certain overhead of shared medium acquisition, usage and release. These times could also be annotated to the proposed simulation model. Effects of instruction and data fetches over the shared communication medium are not covered, assuming that each core has its local data and instruction memory.

Software Shared Objects In this work Shared Objects have only been mappable to Hardware Sockets, which always results in a hardware implementation. As already discussed in the context of multi-tasking above, Shared Objects may also be used for SW/SW or in a mixed SW/SW and HW/SW communication scenario. For this reason, it would be necessary to allow mapping of Shared Objects to the run-time system of a software processor as well. This has been described in [17] with a proof-of-concept implementation on Linux (called OSSS RMI for Linux).

Figure 9.4 depicts the extended interaction and memory layout of client and Shared Objects in hardware and software. To execute a method of such a Software Shared Object, the client has to gain exclusive access to the object's state by obtaining the lock. Secondly, the grant conditions of the method need to be evaluated. When granted, the requested method can be executed. Afterwards, and in case of a non-granted guard condition, the

lock is released again. The explicit implementation of the lock mechanism is required to enable synchronization across core and OS boundaries.

RMI over SystemC TLM 2.0 Currently, OSSS RMI Channels can only be used with OSSS Channels. The main drawback of OSSS Channels is their internal usage of signal level communication. In the worst case, this slows down Virtual Target Architecture model simulation to the speed of an RTL simulation. This speed is insufficient for design space exploration in an acceptable time. For this reason, the OSSS RMI protocol shall also be able to use the SystemC TLM 2.0 API for shared bus communication as well. The RMI channel implementation already allows this adaptation to SystemC TLM 2.0, but it has not been implemented as part of this work. The virtual platform in [17] has successfully implemented the OSSS RMI protocol over a SystemC TLM 2.0 shared bus in Platform Architect, a commercial virtual platform from Synopsys (formerly provided by CoWare).

Simulation speed enhancements Further possibilities to enhance the simulation speed of OSSS, as presented in this work is the consequent usage of `SC_THREAD` processes in Application Layer and Virtual Target Architecture Layer models only. Currently, Actors in the Application Layer model are using `SC_CTHREADS` which are statically sensitive to a clock event. Instead a `SC_THREAD` with dynamic sensitivity and timed event notifications (e.g. `wait(20, SC_MS)`, instead of a series of clock event `wait()` statements will impact the overall simulation speed dramatically. Like RMI over SystemC TLM 2.0, this straight forward optimization has already been applied in [17].

Acknowledgment

First, I would like to thank Ilda and Niko for their patience, understanding, support and care. Without your support, I would not have been able to finish this work. I would also like to thank my parents for their enduring support during my studies and the final phase of this thesis.

I am very happy about the continuous support of my first doctoral adviser Wolfgang Nebel. Thank you very much for accompanying, encouraging and sponsoring me for many years while being part of the OFFIS family.

I am also very grateful for the support of my second adviser Achim Rettberg, the many informal discussions we had and his great encouragement that finally convinced me to carry on and finish this work.

While joining my referee team in the final phase, I received very detailed and precise feedback from my third adviser Rainer Dömer. Thank you very much for taking many hours of carefully reading my work and for traveling from Irvine to Oldenburg to participate in my doctoral defense.

Thank you very much to all members of my examination board, including my three advisers Wolfgang Nebel, Achim Rettberg and Rainer Dömer, the chairman of my examination board Andreas Hein, and Ingo Stierand. I know you all had a tough time reading this very extensive work in a very limited amount of time.

The main results of this work have been obtained in the ICODES project, coordinated by Frank Oppenheimer, my former group leader and director of the Transportation Division at OFFIS. Without your commitment in the final phase of my master's thesis, I would not have considered starting to work at OFFIS. Thank you for always supporting me and being a great mentor for so many years.

Very special thanks goes to the former OFFIS ICODES team members: Thorsten Schubert, Cornelia Grabbe and Claus Brunzema. You have been the best project team I can imagine. Thank you very much for many deep and fruitful technical discussions and for the great technical results we have obtained. Without your particular support, this thesis would not have been possible, since it build on top of many of the technical results you have achieved. In particular, I want to thank:

- Thorsten "Mr. Fossy" Schubert, for pioneering Fossy, the backbone of the OSSS synthesis.
- Cornelia Grabbe, for supporting the conceptual phase of the OSSS Application and Virtual Target Architecture Layer definition, the realization of the Software Tasks and Software Shared Objects and for taking responsibility of the technical coordination of the ICODES project.
- Claus Brunzema, for enabling full C++ support by connecting the EDG C++ front-end with Fossy and enabling reliable regression test.

Another thank goes to my former colleague at the University of Oldenburg, Andreas Schallenberg, who worked in the DFG-funded PolyDyn project. Thank you very much for all your technical contributions to Fossy and your participation in many technical discussions.

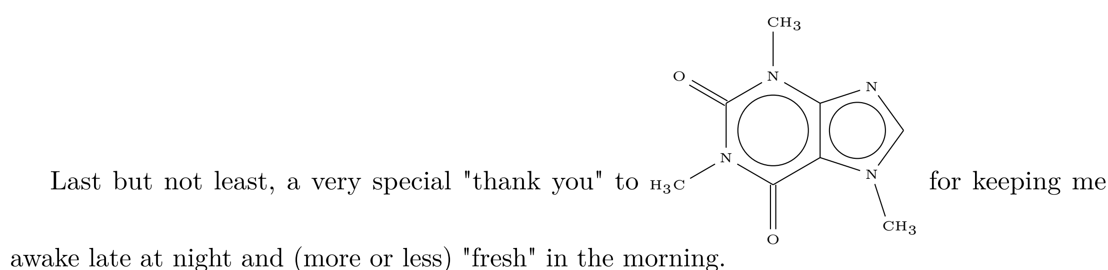
I am very grateful that Philipp A. Hartmann, the best C++ and SystemC expert I know, has joined our team. His exceptional excellent analysis and problem solving skills prevented me from many wrong design decisions. At the same time, Philipp has been the main responsible for enabling software multitasking in OSSS. He has massively influenced all of the OSSS follow-up activities, many of them still running in projects.

The participation of industrial partners in ICODES has also been extremely good, especially during OSSS use-case implementation and evaluation phase. As a result, several evaluation case studies and a substantial survey, which was analysis to set the main goals of this work, have been published. In particular, I want to mention Fabien Colas-Bigey, Anne-Marie Fouilliart, Nico Bannow, Jan Freuer, Alessandro Balboni and Giovanna Ferrara.

Thank you very much to my colleagues Andreas Herrholz, Christian Stehno and Henning Kleen from CoSynth GmbH & Co. KG for picking up the Fossy and OSSS technology, and taking the chance to offer them as a commercial product and using it in commercial design service projects. In particular, I want to thank Henning Kleen for writing an excellent master's thesis about the evaluation of the OSSS RMI communication and for supporting me in the Shared Object synthesis implementation and the evaluation of the synthesis results.

Finally, I want to thank all my current and former group members Maher Ali Fakih, Tiemo Fandrey, Ralph Görgen, Philipp Ittershagen, Daniel Lorenz, Gregor Nitsche, Frank Poppen, Sven Rosinger, Sören Schreiner and Jörg Walter, that I did not explicitly mention before, for their strong support and patience with me over the last years. I really felt to be stuck between a rock and a hard place... but you know what I mean. I am very happy to work with you!

Last but not least, a very special "thank you" to



for keeping me awake late at night and (more or less) "fresh" in the morning.

Kim Grüttner
Oldenburg
March 2015

APPENDIX A

Survey

This survey has been conducted in 2005 to obtain the “Requirements on Hardware/Software Communication Design based on Abstract Communication Models” [96] for the ICODES project [223]. Participants in this survey have been three companies from the automotive, mobile communications and defense & security domain.

Q	Question	Company 1	Company 2	Company 3
1	How does a typical system look like? What are typical components of your system?	SMC products are typically the Node Entities of the Radio Access Networks (e.g. BSC, BTS, TRAU for GSM). In the following we will consider as system the sub-system constituted by a PCB.	Inside Company 2, different applications of completely different complexity are being developed. Within the context of ICODES Company 2 will be present with a video system that can be found at the upper range of complexity. Such video systems in general consist of different standard and non-standard processors (FPGA, DSP, μC), standard memories and discrete logic on a electronic control unit (ECU). The connection between the components is mainly done by standard buses. Due to the fact that we use μC instead of μP, there is usually a subsystem provided with the processor core, all connected with a single or several buses.	In Software Radio Applications, the equipment has to support real-time base-band processing, protocol stacks implementation, management and control. Presently, the HW architecture is mainly heterogeneous and includes several DSPs, GPPs, ad FPGAs. For hand-held terminals, such processors are integrated inside System-on-a-chip (SoC) or Network-On-a-Chip (NoC) systems (e.g. OMAP2 from Texas Instruments)
2	What is the complexity of the system in terms of development time?	The average time is 1 year.	The development time is very different in meaning of different development stages. The development follows the scheme of different samples A (functional) – B (trial) – C (release) – D (pre-production, 1st sample) – series/mass production. From the start of a series development (assuming all fundamentals are clarified) till start of series production 1,5 years is a good number.	The development time depends on the complexity of the system and on team size. Generally, it takes between six months and one year.
3	What is the ratio between hardware and software development time?	It is design dependent; e.g. in DSP-based system SW development can take 70% of design effort and time; in other designs (data manipulation and control systems) the ratio can be the opposite one.	This is absolutely different, even within the same application but several system generations. Depending on the video system, the sample and generation we have systems with HW:SW 10:90, 50:50 and 100:0 rate.	Of course it depends on the design; for Digital Signal Processing systems the ratio is roughly 60 % SW and 40 % HW.
4	In which language (C++, SystemC, English, Matlab, ...) do you start describing your typical application?	Natural language is used to describe system specification. Design subsets or algorithms are initially described using Matlab and C/C++. Tools like MLDesigner or Cadence SPW (now Coware) are sometimes used to simulate communication paths (transmitter/receiver).	Discussions of the system requirements with the customer are often based on natural language documents (english, german). For the system implementation process an abstract (executable) model (C/C++, Matlab/Simulink, Labview) is often used as a starting point.	First specifications and requirements are often written in natural language documents. In the DJD document, Dossier Justificatif de Definition, use cases are now more and more described using UML language. • For Signal processing algorithm Matlab or C/C++ language • For Hardware natural language, • For SW natural language and UML SystemC is under evaluation process for HW and SW modelling.

continued on next page

Q	Question	Company 1	Company 2	Company 3
5	What are these models needed for?	Mainly for functional simulation and to verify proprietary algorithms.	They are used for describing and evaluating different algorithms (functional exploration and verification) and rapid prototyping purposes. They are further used to describe the data flow, provide performance estimation or can be used as reference design or documentation. Furthermore, the purpose of such models is to have a clear view of a design and get an early estimation of the system's behaviour. The introduction of SystemC is currently under evaluation because this methodology/language promises all above named advantages in a practicable way. C/C++ as a typical modelling language is often being moved into the series SW development after prototyping.	These models are used to explore and validate DSP algorithms. Test benches can be delivered by the Signal Processing algorithm developer to the HW designer to validate the final implementation.
6	Do you make pre decisions about hardware/-software partitioning? Which pre-conditions are deciding?	HW/SW partitions are mainly influenced by application specific components availability. Costs and performance analysis are made to define which key components will be more suitable for the system and then the HW/SW portion is strongly influenced by that choice. Decisions are taken trying to define which will be the most effective implementation for any specific function needed to implement the overall system. The need of flexibility or uncertain specifications are driving factors towards SW implementation. Currently there is not a tool supported decision process.	By assumptions, by experience, by calculating throughput/capacity/etc., by testing different algorithms, by required design time + costs for implementation. Decisions are often based on predecessor systems that are already available (in particular if the system is an extension or further development of an existing one).	Functional and non-functional criteria are used to make pre decisions about HW/SW partitioning. They are mainly derived from expert advice. • Functional: time consumed, throughput, etc. of a resource. Need of reconfigurabilty of the system • Non-functional: power consumption, cost, overall dimension of the system, ...
7	How important are IP blocks?	IP block are mandatory. As the pressure to reduce the design phase is growing higher and higher, the use of IP functions is one of the viable ways to address the timing pressure. In order to shorten the design cycle, IP function blocks are usually purchased from a 3rd party company and not developed in house.	IP blocks are very important from a reuse point of view and due to the fact that third party components (like μCs) are used. Whenever possible we try to reuse existing IP (like standard filters for video manipulation) or if costs are acceptable, be buy IP. But using IP depends on the availability of IP for a certain function / algorithms. Especially advanced algorithms often do not exist as IP and therefore need to be implemented from scratch.	IP block are important for reuse because system complexity increase and we have to face time to market and development cost. Sometimes we are induced to use IP for compliance, for instance because of standard interconnection (PCI).

continued on next page

Q	Question	Company 1	Company 2	Company 3
8	What are typical IP blocks?	Coherently with the goal to shorten the design cycle, usually IPs are mainly related to the implementation of 'standard functions' or 'standard interfaces'. Typical IPs are e.g. GMAC block, UTOPIA I/F, SERDES, PCI... In the last period an important class of IPs has been adopted in Company 1 design process: microprocessors embedded in FPGA devices. Focusing our attention only to the market leader companies, example of those IPs are PowerPC 405c and MicroBlaze (available in Xilinx devices), ARM and Nios (available in Altera devices) and all the peripherals related to such components.	μCs, standard μC peripherals, standard blocks like ADC, DAC, Generic memory (FPGA), Memory-IF, Filter, Multiplier, Bus-IF, Standard SW like OS and standard functions, etc.	• DSP blocks as: FFT, FIR, NCO, Reed Solomon, Turbo Codec, Viterbi, ... • Bus as RapidIO, HT • Processors and microcontrollers, • Memories (SDRAM), • Networking IP as Utopia, Ethernet MAC, stsfrm, EDK 6.2, etc. • We use either Altar and Xilinx devices
9	Could you give an estimation about the percentage of existing IP with respect to the overall design?	The percentage of current design implemented using 3rd party IPs can obviously vary depending on the specific design needs. A rough estimation starts from no IPs to 60% of a whole design, implemented by IPs.	From 0-70% on ECU's depending on the design. Up to 90% in the ASIC development process, if regarding "fine grain" reuse (which means reusing and adapting blocks from former designs)	Use of IP blocks can vary depending on the system. Sometimes it can be 90 % but the average is between 20 and 30 %. Nevertheless, we can see that the trends for using IP blocks is increasing.
10	What kind of interface does a typical IP-Block have?	Excluding Microprocessors, currently adopted IPs interact with other design functions via standard interfaces and busses and protocols. E.g. GMII I/F, Utopia bus, ATM protocol, Ethernet protocol etc ....Microprocessors have their own proprietary busses (like AMBA or OPB) but, especially using microprocessor soft core (like Xilinx's MicroBlaze and Altera's Nios) it is also possible to interact with the HW portion of a design in a very direct manner using shared memories or FIFOs. The design environments proposed by Altera and Xilinx offer (or will offer soon) some utilities to implement 'direct' interfaces to HW.	Standard buses, specialized IF, propriety buses, C++ APIs, automotive protocols (CAN, LIN, FlexRay, ...)	Direct interfaces, Interfaces as FIFO, memories, registers, Standard buses as AMBA, Proprietary buses

continued on next page

Q	Question	Company 1	Company 2	Company 3
11	On which abstraction level do you integrate the IP-Blocks into the simulation of the whole system?	We do not simulate the system (PCB). Simulation is carried on at subsystem level (ASIC-FPGA-PlatformFPGA-Embedded processors). The IP blocks in general are available as black-boxes. Usually the IPs are integrated at RTL level. For the RTL portion we use the synthesizable subset of VHDL. The description of the sub-system in which the device under development will operate at a higher level of abstraction. We use the whole expressivity capacity of VHDL and Verilog and also manage cosimulation with block described with Systemc 2.x . Other simulation oriented IPs are used to create a simulation scenario. For the early stage functional simulation we always look for a fast and functional accurate model .We could define it as cycle accurate as, in this phase, we mainly perform functional simulation in a clocked environment. As first trial we require an IP-Block behavioural model to the company from which we receive the IP-Block and in case of non-availability of this kind of model we currently generate in house a functional behavioural model usually described in VHDL.	Mainly on all levels. Usually there is a high level model description (e.g. for Matlab) as well as a C++ and/or VHDL description. ASIC design flow: VHDL at RT-level	Usually IPs are integrated at RTL level, we currently us VHDL RTL or netlists.
12	Do the IP-Blocks have standardized Interfaces (AMBA, ...)?	Mostly. See answer nr. 10	Partially yes in case of automotive busses (CAN, LIN, FlexRay, ...). Furthermore, only in very few parts (e.g. PCI, Memory IF)	As said in 10 we sometimes use proprietary buses.
13	Are you using SystemC or are you planning to do so?	At the moment SystemC 2.0 has not been used in the productive design flow but we run some experiments and we are planning its adoption in the next future mainly as functional modelling description language, for functional verification and to explore different architectural hypothesis.	Not yet for commercial development but for evaluation. We are trying to establish SystemC in our development because of the advantages, it promises. However, applicability in industrial area has to be proven. ASIC design flow: first steps of introducing SystemC currently done: SystemC for testbench support (SystemC verification library)	SystemC is in an evaluation process. Applicability to industrial process has been proven, we are now using it in "pilot projects" in parallel with standard designs.

continued on next page

Q	Question	Company 1	Company 2	Company 3
14	Do you create transaction level models? What are these models needed for?	In some experimental trials we created TLMs only with functional verification purposes.	Yes. We use transaction level models for abstract modelling of function calls and data transfers between modules. Transactional models can be fully untimed or the can be provided with a minimum of timing information like the overall execution time of a complete transaction. This models do have the advantage of fast implementation and high simulation speed if an accurate timing modelling is not required.	We are now developing transactional level models of HW IP blocks or we use models done by the IP provider. We also use un-timed transactional level to model Signal Processing algorithms and do the refinement step from floating point to fixed point representation before their coding towards a specific DSP or in a HW block.
15	Do you use SystemC's (hierarchical) channels?	In the experimental trials mentioned in the two answers above we used some hierarchical channels. We are in fact interested in the capability of modelling channels which may contain quite complex behaviour as opposite of primitive channels (e.g. signals) which cannot contain internal structure.	We are using hierarchical as well as primitive channels but prefer hierarchical channels when the channel has to work in a "call → exec → return" manner with blocking access. This is for complex data transfers often the easiest way of implementation.	We use Primitive channels as FIFO and mutexes. We have done some exercises with hierarchical channels
16	What kind of modelling concepts would you prefer in OSSS, which you know from languages like VHDL, C++, SystemC,...? And why?	In general we would like to maintain all the functional modelling capabilities of traditional hardware description languages. We do not expect to have all the C++ and SystemC 2.0 concepts but we expect to take advantage of OO concepts as inheritance and of smarter communication concepts.	RTL (keep the chance of an 1:1 design, only described by a designer, not compiled by a behavioural tool, keep compatibility with existing RTL IP, keep the chance to use formal verification tools for 100% verification of source code with compiled netlist), Behavioural (for faster modelling), templates/generics are absolutely required, recursive function calls (with constant number of iterations) are sometimes used within VHDL for design creation, in VHDL generate constructs are very often used ï¿$\frac{1}{2}$ for SystemC there is no real concept for this (only a array of references to one module and a loop that creates several module instances.	More and more, we have to model large systems at various levels and re-usability, portability capabilities become important issues. So, it means for us that having an "object oriented" way of designing our applications is crucial. Inheritance and polymorphism are concepts which have to be supported.

continued on next page

Q	Question	Company 1	Company 2	Company 3
17	Where do you see the advantages of each of the different modelling languages?	C++ (or C): in simulation because it is very fast in particular for high abstraction level simulation VHDL advantages: in HW modelling phase because it is standard, it is well known by hw designers, it is widely supported by EDA vendors, it allows reuse, it allows technology independence SystemC: in system modelling phase because it is a common language for both HW and SW; in addition to this it is open source, it is standard…	• Matlab/Simulink: Conception phase for HW/SW independent system modelling, heavily used in automotive due to highly heterogeneous systems. • VHDL and Verilog: HW design modelling (established methodology and tools, full control of the generated design by a designer, reuse of existing IP, some tools allow to compile a design under test into an FPGA and integrated this design into the existing testbench that offers advantages in performance and gives first impression of the real design behaviour), mainly RTL • C++, testbenches can be created and maintained very well and provide a lot of options to inject and capture data in a simple way. C++ models can be compiled to an executable model and do not run inside a simulator. C++ has significant advantages in performance vs. VHDL simulators. It is easy to provide a compiled C++ design hiding it's implementation (if source code is not provided)	• C/C++: It is natural language used by algorithm developers, simulation are very fast • Matlab: For signal processing application it is an easy way to rapidly build a system, simulate and visualise the results. It is also like a standard to exchange part of an application between different teams. • VHDL: It is the standard for HW developers and widely supported be EDA tools. • SystemC: It is a unique language for functional modelisation of mixed HW/SW system level down to TLM and cycle accurate simulation. It implements object oriented techniques.
18	How would you like to model hardware and software in one language?	Our main desiderata are the capability to describe a subsystem functionality using a single description language without having 'a priori' constraints about the partition between hardware and software portion. The abstraction level could be timed or untimed depending on the design needs. We would like to have the ability to use this model to help the designer decision about hardware/software partitioning. After the definition of the target platform (processor and HW resources) we would like to have at least useful hints to refine SW and HW parts to fulfill the target design environment requirements.	SystemC seems to be a good candidate but has currently some disadvantages (not all VHDL features are implemented yet a methodology for generating loops is not yet defined). Currently there are only very few synthesis tools available. Furthermore, these synthesis tools usually have major restrictions. On the other hand there is no common language to model HW/SW (e.g. UML is used for abstraction and flow, but not for implementation)	To use only one modelling language is very important, because we like to be able to model a system at various level of abstraction, also be able to do partitioning and mapping of the application on a specific architecture, and in a "spiral" and incremental process be able to converge to the best solution (implementation) by mean of simulations. Such kind of development is very hard to do when you have several languages together.

continued on next page

Q	Question	Company 1	Company 2	Company 3
19	How would you like to model the communication and the synchronization between different components of the system?	We would like to have specific constructs to ease the design process. Both HW and SW designers should be able to manage coherently the interface without a deep knowledge of both worlds.	The modelling approach of communication and synchronization between modules has to be flexible enough to cover transactional, cycle accurate, bit/signal-accurate, RTL level, master/slave-communication, etc.	We would like to model communications at each level of abstraction in order to be able to do refinement in expanding an abstract communication protocol into an implementation. We want first to address Transactional and cycle accurate level modelisation.
20	Which not[at]ion of time is of interest?	For description and analysis, the lower level of timing abstraction is the clock cycle. We would like anyway to be able to use untimed modelling where timing notation is not essential to capture the functionality.	The notation of time has to be flexible enough to cover untimed transactional, timed (event triggered (clock/signal)), delay like wait(100, SC_MS), function/module execution time, cycle-accurate, bit/signal-accurate	Clock cycle, transactions
21	What means object-oriented HW/SW design to you?	Up to now we had not yet considered object-oriented HW-SW design techniques	Using classes, object instances, inheritance, polymorphism, virtual interfaces (templates are a C++ feature but not real object orientation, because it is a pre-compiler issue) Modelling features outlined above plus analysis and synthesis capabilities (Link to implementation required!)	This means easy use of components because blocks can be changed without concern for their implementation, assuming that interactions are well defined.
22	Do you employ any object-oriented techniques during the design process?	No	In SW yes, in HW not yet	Object oriented techniques are used in SW development but not for the moment in HW design.
23	If so, which techniques do you use and where in the design process do you use them?	Not applicable	See 22, furthermore: we use C++ including real object oriented design features from concept until series code.	-
24	What are their benefits?	Not applicable	Reuse, creation of libraries and IP, encapsulation, reduced development time, less error-prone (See standard arguments for OO software design)	-
25	What is the complexity of the software in terms of lines of code, number of tasks, amount of memory, ...?	Focusing on our target technology (Platform FPGA), we have in general applications of some thousands of lines of code (1000-8000). In some cases this code is in part developed from scratch and in part inherited from previous releases of the applications. Often there is only one task with interrupt based handling of the functions. Memory is extremely project related. It spans from 500Kbits to some Mbits.	This absolutely differs between different functions. Furthermore, the used OS on SW should be added to the complexity of a design. Usually there are several tasks running in parallel in SW and HW. The memory can be a few bytes up to a few MBytes.	This depends on the applications of course. In our communication systems most of our Software work-packages are 50-200 K SL.

continued on next page

Q	Question	Company 1	Company 2	Company 3
26	How important is multitasking?	Focusing on our target technology (Platform FPGA), as far as we concern, multitasking is not mandatory at the beginning. In general we deal with the implementation of several functions on a platform which has one or more processors and millions logic gates available. In principle, any function could be implemented using several 'simple' processors performing a single task without an OS on board. Alternatively we could use a more complex processor with a RTOS to perform the tasks management. Of course the design needs will drive the technical choice. In general we can start with the easier architecture and add multitasking in a subsequent phase if possible.	Due to the fact that we use a real time OS, with several threads running on it, multithreading is a must in SW partitions. In HW any process is running in parallel. Therefore we can say that we have several threads running in parallel. We are using processes in HW+SW that are working on completely different tasks/issues beside the 'low level' processes that e.g. 'just' synchronize some input data to the internal clock.	Multitasking is more and more use, it is important to manage time in complex applications. Moreover, applications are becoming more and more "dynamic" with tasks running in parallel on different HW resources.
27	Does the software have real-time constraints?	Yes. Our designs are in the infrastructure telecom domain. We surely have timing constraints for the design portion related to voice management and so we can consider them as real-time constraints. We can have more relaxed timing constraints for design related to data management.	We are using a real time OS and we definitely need the real time capability to schedule different jobs/tasks by setting real time constraints. So the answer is yes.	Because we design communication applications, hard real-time constraints are present at baseband processing. Soft real-time for protocol implementation
28	What is/are your current implementation language(s) for the software?	Focusing the answer on Platform FPGAs, the software implementation language is strictly constrained by the design environment requirements set by the chosen 'FPGA foundry'. In general we use 'C/C++' or a dialect of 'C/C++' that means 'C/C++' plus special procedures and libraries available in the chosen design environment and necessary to interact with the specific target. Usually we use specialized GNU compilers enhanced to exploit Platform FPGA characteristics.	C and C++ plus a few lines of assembler inline code (directly added into the C/C++ source code or encapsulated in C/C++ functions/libraries). In the automotive area synchronous languages like the imperative language ESTEREL and the data-flow language LUSTRE are also important.	We are using C (sometimes C++) language and sometimes assembly code for inner loops in signal processing applications on the DSP side.
29	What is the complexity of the hardware in terms of lines of code, number of entities, number of gates, ...?	Typical FPGA based HW: from 100 to 130 VHDL files and consequently entities; each file from 50 to 2k lines; from 10k to 30K FFs, from 15k to 45K LUTs.	An gate equivalent from a few 1000 gates to a several 100k gates, depending on the application.	For complex FPGA we can have 10 000 lines of code (in several files), it depends on the design and its nature. For Signal processing we have less lines of code for more gates and for control and interconnections it is the opposite.

continued on next page

Q	Question	Company 1	Company 2	Company 3
30	What is/are your current implementation language(s) for the hardware?	VHDL for RTL level and Verilog for Netlist (gate level) description.	VHDL and very few lines of Verilog (as source code or netlist) if IP is being used.	We are using VHDL and very rarely, when using IPs, Verilog
31	Do you perform any hardware/software cosimulation? How?	Targeting Platform FPGAs, in our first trials we use a development board (commercial or house-designed) to perform some verification for hw-sw design. This approach was possible due to the small dimension of SW portion of the design. At the moment we haven't yet defined a well-structured method and tool set to perform HW/SW cosimulation even if we are aware that commercial environments are available (Seamless VirtexII Pro cosimulation package)	Only for evaluation purpose: up to now by proprietary, hidden tool implementations of design tools or by simulation tools with an open interface like ModelSim. For HW prototyping using vendor specific platforms like ARM Developer Suite, etc.	For verification we use development board. For platform, with ARM processor for example, can use Seamless tool for cosimulation.
32	Which constraints/conditions are essential to find a suitable HW/SW partitioning?	Possibility to evaluate from the early phases the feasibility in term of memory occupation, hw resources used, timing constraints, cost of interconnections.	Area, speed, effort for implementation, flexibility, existing IP, available components (like processors). The most important point is: keep the costs as low as possible (of course requirements have to be fulfilled) and take respect of the customers presets (μC to be used etc.)	To find a suitable partitioning with have to take into account: • Time constraints, • Processing efficiency • Flexibility • Power consumption
33	At which phase of the design flow are you planning to debug the design (before or after the high-level synthesis)?	SW debugging and HW verification take place several time at different design phases. It's a continuous process In our opinion we should plan a functional debug/verification as soon as possible in the design flow, consequently also before any synthesis task. The debug/verify process has to be performed in every design phase, with an increasing detail level and with appropriate tools (not always available today)	Algorithmic verification with tools like Matlab usually starts before C++ implementation. The debugging of the design is accompanied by the implementation (if a piece of code has been edited, it will be simulated). Unfortunately it is very hard to simulate the complete design while parts are ready only. Furthermore, debugging of real time systems can become really complicated. In general , verification/debugging should be possible as soon as possible to support an early (model-based) integration test.	Debug of the design takes place at each step of the design flow. It a an iterative process. We have to debug before the synthesis step.

continued on next page

Q	Question	Company 1	Company 2	Company 3
34	Which language should be supported from the high level synthesis for the hardware?	VHDL	If the question refers to the importance of the output format of high level synthesis the answer is: VHDL (is a must), SystemC (might become very important, once SystemC synthesis tools become more popular, might be important for behavioural synthesis tools. Furthermore SystemC output can easily be used inside the same testbench as the high level design description) If the question refers to the importance of the input format for high level synthesis the answer is: OSSS in addition to above mentioned VHDL / SystemC	If we are speaking of the output of the high level synthesis it is VHDL.
35	Which language should be supported from the high level synthesis for the software?	C/C++	Synthesizable subset of SystemC, C/C++ constructs inside synthesizable SystemC construct	If we are speaking of the output of the high level synthesis it is C/C++ or embedded C.
36	How important is it for you to get readable generated output (VHDL, SystemC, ...) out of the high level synthesis result? Why?	It is important in particular when you have to solve timing problems, assigning specific constraints on some particular objects before synthesis or after it or during floorplanning. If the VHDL RTL is not readable, the generated netlist will be completely cryptic: it will be almost impossible to insert special constraints. When you need a sophisticated and accurate manipulation of the backend netlist to solve complex implementation problems, you need a traceable input.	The importance inside Company 2 is from important to very important! On a range from 0 (low) to 10 (high) it would be a 7 to 9. The reason is to have an option for a designer to find a bug if synthesis doesn't produce the expected results (e.g. having in mind that every tool has some bugs and furthermore a designer might cause a wrong synthesis by wrong coding style or setting wrong synthesis parameters. This all might create defect designs that need to be debugged to find the problem in order to fix it). Thus debugging of intermediate design steps is required. Therefore, the more the code is readable, the better it is.	It is important for debug if any problem occurs and for timing constraints. We do not want to modify this generated code.

continued on next page

Q	Question	Company 1	Company 2	Company 3
37	What kind of synthesis tools and compilers are you planning to use after the high level synthesis?	Synthesis tool: SynplifyPro by Synplicity. Compilers: Xilinx Platform FPGA SW flow is based on enhanced versions of GNU Tools targeted on MicroBlaze and PowerPC: mb-gcc compiler, mb-as assembler and mb-ld loader/linker; powerpc-eabi-gcc compiler, powerpc-eabi-as assembler and the powerpc-eabi-ld linker. Some options have been added or enhanced in the GNU tool suite for the EDK. Both the compilers (powerpc-eabi-gcc and mb-gcc) use certain libraries for all the program, and in particular the one which contains drivers, software services (such as XilNet & XilMFS) and initialization files developed for the EDK tools. he microblaze compiler options new with respect to GNU standard compiler are related to the possibility of exploit particular characteristics of the targeted processor, microblaze or ppc (for example the choice to use hw multiplier or not) or to specific on-board debugging operations. The PowerPC GNU compiler (powerpc-eabi-gcc) is built using the GNU gcc version 2.95.3-4. No enhancements have been done to the compiler. The PowerPC compiler does not support any special options. All the listed common options are supported by the powerpc-eabi compiler. Currently the team experience refers mainly to Xilinx but the Altera environment is comparable. the sw Tool Chain is based on the standard GNU C compiler (GCC) compiler, assembler, linker, and makefile facilities.	VHDL: Synopsys dc_compiler, but the created output should be generally valid and not restricted to a special tool. C++: any standard C++ compiler	We are using for FPGA design: • Summit VisualHDL : 6.7 • Summit VisualElite : 3.5 • Textual editors like grasp, ultraedit, emacs • Cadence NCSim ldv 4.1, ldv5.0 • SynplifyPro 7.6 • Mentor Graphics ModelSim 5.8 • Mentor Graphics Leonardo Spectrum2003.a , Precision2004.a • Altera Maxplus2 10.2 • Altera Quartus2 4.0 • XilinxISE m6.2i We are using Design Complier for ASIC. For software we are using any C and C++ compiler.

continued on next page

Q	Question	Company 1	Company 2	Company 3
38	How do you model/realize the communication and synchronization in a hardware/software design up to know?	Up to now, for the Platform FPGA target technology, due to the lack of automation we relied upon the limited methodology suggested by the component suppliers E.g. in the EDK environment you describe the HW system via the mhs file, a text based representation of the system in which you add the selected bus, the predefined peripherals and your code as new IP: You have predefined drivers and you are guided in creating your new drivers for your code. Our experience is limited to simple memory mapping or interrupt handling approaches.	By abstract IP blocks it is hidden (like if blocks are designed for Matlab for example), on RTL level (e.g. existing IP) on signal sensitivity, on SW by semaphores, on HW+SW by interrupts, shared memory, direct communication, communication via DMA	We realise communications and synchronisation in HW/SW design by: • Interrupt mechanisms, • flags • polling • shared memories, DMA • direct HW/HW link (signal) • drivers
39	Which output of the high level synthesis do you need for the interfaces between hardware and software?	Automatic generation of VHDL modules and C/C++ drivers	On HW side: An IF of the module to a specific bus (is a must) and a specific bus description (would be very nice but not required). On SW: A piece of SW that can be integrated into the existing SW environment + source code. This might include an import into a SW design with or without an OS. Furthermore, all information of the interrupt vector table (with/without OS) or a thread wrapper or even a linkable driver (for an OS) should be provided. Of importance is that existing standards should be regarded.	On SW side we would like to have VHDL RTL and on SW we would like to have code to be integrated, C/C++ drivers.
40	What are the targets for the hardware and the software: FPGAs, CPUs, ...?	Platform FPGA with soft core processors (microblaze, nios) and hard core processors (PPC405c, ARM). Preferred solution is the Xilinx Virtex Architecture with PPC	FPGA (Xilinx Vertex: indeed it is not required to provide Xilinx specific modules, blocks, etc. Instead the code should be generally valid that can be compiled to any target architecture. The code should not contain target specific parts like CLOCK_DLL or I_BUF_G, Blockram or similar issues), ASIC (general), μC (PPC603)	We are using • FPGA Xilinx or Altera • Hard core as ARM, MIPS • RISC CPU
41	Which operating systems do you use for the target system? none or ?	None. If needed we prefer an Open Source Solution (e.g. UCLinux)	OSEK, VxWorks, RTOS-Linux	We are using Vxworks , Lynx-Os, ARTK

continued on next page

Q	Question	Company 1	Company 2	Company 3
42	How does your interface between hardware and software and vice versa look like?	In general our applications use data structured as packets with need of parallel manipulation and processing. Nevertheless we are moving towards fast serial links. We are interested in using all the state-of-the-art technologies including RocketIO, PCI Express, etc. to reduce interconnection complexity at system level. We use all the traditional HW structures as buffers, registers, FIFOs, addressable memories, using standard protocols when interfacing commercial components and ad-hoc protocols within the subsystem. See also answer 38.	• Direct access to memory and peripherals by writing to / reading from addresses • Preparation of data inside memory (in C/C++ as variables and objects or by writing directly into memory) and data transfer by DMA • Active and passive transfer (active write / being read) • Interrupts / Polling	See 38.
43	If you use bus systems, which bus do you mostly use?	OPB and PLB bus	Currently used buses are PCI, several different serial LVDS protocols, CAN, LIN, FlexRay, MOST and further. The used buses however are in a steady process of permanent evolution, adaptation and replacement caused by the steadily changing requirements like performance, flexibility, safety, reliability. Therefore, the bus description should stay as much flexible as possible (if possible) or in other word: During the context of ICODES, it would be an emphasized requirement to provide an API, into which the designer can place an own bus model that would allow an easy replacement of different bus descriptions.	Amba bus, FPGA Core Connect, VME, proprietary buses

continued on next page

Q	Question	Company 1	Company 2	Company 3
44	What do you think is the greatest bottleneck in the design flow that should be addressed in the context of ICODES?	We consider as the main bottleneck in the design flow, the gap between system level specifications and HW or SW flows, i.e. the lack of a well defined system level starting description in the supported foundries flow and the non-automated operations required no flexible exploration of HW-SW partitioning. In addition to this there are some other problems and areas of improvement in the design flow: the existence of unconnected HW-SW development flows, the poor co-verification flow and tools especially in the earlier flow steps, the difficult initial analysis and identification of problems in the very early steps of the design flow, the lack of "high level models" (e.g. ATM Workbench), the crucial issue regarding inter-modules communication description HW-HW, HW-SW, SW-HW, the poor support of team-based design flow.	Interaction of different tools / description languages • In conception/exploration phase: Fast evaluation of different design alternatives • In implementation phase: In HW/SW: easy modelling of communication, smooth link into implementation flows on HW and SW side as well High level object oriented, synthesizable HW description currently is a not existing part in the design flow. However, the designer should be able to keep influence on the created design to have full control of the results if required like cycle accuracy, RTL description, created memory/register structures	Either in SW or HW design communications between IPs (HW or SW) are becoming more and more complex. Having the ability to model and synthesize those communications among those different IP could bring a real plus on the quality of the design (because the HW or SW communication code is generated and so good "by construction") and productivity (improved by code generation techniques).
45	How does your envisioned solution for this problem look like?	We think that the directions in which the research has to address its effort. is the exploitation of the high level approaches and languages together with the development of closer HW and SW design flows, and of high level analysis tools (what-if analysis, performance analysis,...) to help HW-SW partitioning and architectural choices. In addition to this the application of metrics is surely of interest. Finally, when it is possible, it could be very useful the reuse (with refinements) of high level descriptions as much as possible, using appropriate synthesis tools.	Currently we are focusing a methodology that allows to break a C++ design description into a executable SystemC system. description. Timing (execution time that simulates data transfers or is similar to the original C++ function execution time) can be easily inserted. Deadlines can be watched for. HW/SW modules can be easily replaced vs. each other, buses can be replaced, modified or new structured. The result is able to gain a fast estimation of how a system will behave. Furthermore different system configurations can be evaluated within a minimum of time. Nevertheless, this solution is mainly thought for system simulation and creation of design structures but has not yet a focus on synthesis. Link to synthesis is strongly envisioned.	-
46	Where the questions in this document clear to you?	Some questions will become clearer and some answers more detailed as part of the ICODES activities when our team will gain more experience on integrated HW/SW interconnection design and synthesis	In general yes	-

APPENDIX B

Timed Automata Templates and Examples

B.1 Used scheduling algorithms

Uppaal implementations of the scheduling algorithms described in Section 5.5.3.

These scheduling algorithms are used inside the Shared Object's Arbiter timed automaton template as described in Section 5.5.6 as well as in the Shared Bus Scheduler timed automaton template as described in Section 5.6.4.

```
1  // initialisation for Round Robin & Modified Round Robin Scheduler
2  index_type last_grant ← NC−1;
3
4  // history initialisation for Ceiling Priority & Least Recently Used Scheduler
5  index_type history[NC];
6  void initialise_history () {
7     for(i : int [0,NC−1])
8        history[i] ← i;
9  }
10
11 /**************************************************
12  *   Static Priority Scheduler
13  **************************************************/
14 index_type schedule_static_priority(bool zero_is_highest) {
15    if (zero_is_highest == true) { // zero is highest priority
16       for(i : int [0,NC−1]) {
17          if(mid_guarded[i] != NOP)
18             return i;
19       }
20    }
21    else { // zero is lowest priority
22       for(i : int [0,NC−1]) {
23          if(mid_guarded[NC−1−i] != NOP)
24             return NC−1−i;
25       }
26    }
27    return −1;
28 }
29
30 /**************************************************
31  *   Ceiling Priority Scheduler
32  **************************************************/
33 index_type schedule_ceiling_priority() {
34    index_type grant ← 0;
35    bool ripple ← false;
36    for (i : int [0,NC−1]) {
37       if (mid_guarded[history[i]] != NOP ∧ !ripple) {
38          grant ← history[i];
```

```
39             ripple ← true;
40           }
41           if (ripple ∧ (i != NC−1)) {
42             index_type swap ← history[i];
43             history[i] ← history[i + 1];
44             history[i + 1] ← swap;
45           }
46     }
47     if (ripple)
48         return grant;
49     else
50         return −1;
51 }
52
53 /*************************************************
54  *    Round Robin Scheduler
55  *************************************************/
56 index_type schedule_round_robin() {
57     index_type nextGrant ← last_grant;
58     bool breakFlag ← false;
59
60     for(i : int [0,NC−1]) {
61         if (!breakFlag) {
62             if (nextGrant == (NC−1)) {
63                 nextGrant ← 0;
64             } else {
65                 nextGrant ← nextGrant + 1;
66             }
67             if (mid_guarded[nextGrant] != NOP) {
68                 breakFlag ← true;
69             }
70         }
71     }
72     if (breakFlag) {
73         last_grant ← nextGrant;
74         return nextGrant;
75     }
76     else
77         return −1;
78 }
79
80 /*************************************************
81  *    Modified Round Robin Scheduler
82  *************************************************/
83 index_type schedule_modified_round_robin() {
84     index_type nextGrant ← last_grant;
85     bool breakFlag ← false;
86
87     for (i : int [0,NC−1]) {
88         if (!breakFlag) {
89             if (nextGrant == (NC−1)) {
90                 nextGrant ← 0;
91             } else {
92                 nextGrant ← nextGrant + 1;
93             }
94             if (mid_guarded[nextGrant] != NOP) {
95                 breakFlag ← true;
96             }
97         }
98     }
99     if (breakFlag) {
100        if( last_grant != nextGrant ) {
101            if (last_grant == (NC−1)) {
102                last_grant ← 0;
103            } else {
104                last_grant ← last_grant + 1;
105            }
106        }
107        return nextGrant;
108    }
109    else
```

```
110         return −1;
111 }
112
113 /**************************************************
114  *   Least Recently Used Scheduler
115  **************************************************/
116 index_type schedule_least_recently_used() {
117     index_type grant ← 0;
118     index_type j;
119     for(i : int [0,NC−1]) {
120         if(mid_guarded[history[i]] != NOP) {
121             grant ← history[i];
122             for(j ← i; j < NC−1; j++) {
123                 history[j] ← history[j+1];
124             }
125             history[NC−1] ← grant;
126             return grant;
127         }
128     }
129     return −1;
130 }
```

Listing B.1: Scheduling algorithms from Section 5.5.3 modeled in Uppaal

B.2 Application Layer TA example

```
1  // definition of timing annotation intervals
2  typedef int[0,1] EET_type;
3  const EET_type BCET ← 0; // lower bound: Best−Case Execution Time (BCET)
4  const EET_type WCET ← 1; // upper bound: Worst−Case Execution Time (WCET)
5
6  // number of client processes
7  const int NC ← 2;
8  typedef int[0,NC−1] client_type;
9
10 // definition of method ID types
11 typedef int[0,2] method_type;
12 const method_type NOP ← 0; // No OPeration (NOP) represents no service call
13 const method_type PUT ← 1; // represents the put service
14 const method_type GET ← 2; // represents the get service
15
16 // definition of status ID types
17 typedef int[0,2] status_type;
18 const status_type WAIT ← 0;       // waiting for access to SO
19 const status_type GRANTED ← 1;    // access to SO granted
20 const status_type COMPLETED ← 2;  // access to SO has been completed
21
22 // index type for Shared Object scheduler
23 typedef int[−1,NC−1] index_type;
24 index_type granted_cid ← −1; // initially no client is granted
25
26 // definition of FIFO size
27 const int FIFO_SIZE ← 5;
```

Listing B.2: Global definitions

```
1  // Timing annotations, written as [BCET, WCET] intervals
2  int put_EET_before[2] ← {10, 15};
3  int put_EET_after[2]  ← {4, 5};
4
5  int get_EET_before[2] ← {10, 15};
6  int get_EET_after[2]  ← {4, 5};
7
8  int EET_eval[2] ← {1, 1};
9  int EET_sched[2] ← {1, 1};
10
11 int[0,FIFO_SIZE] num_elements ← 0;
12 int EET_service[2][2] ← {{2, 5}, {2, 5}};
```

```
13
14 // Timing analysis:
15 // global clock
16 clock t;
17 // timing requirements for put and get clients
18 const int PUT_PERIOD ← 55;
19 const int GET_PERIOD ← 55;
20
21 method_type mid_request[NC];
22 method_type mid_request_guarded[NC];
23 method_type so_mid ← NOP;
24
25 status_type call_status[NC];
26
27 urgent chan put_call, put_ret;
28 client_type put_cid;
29 method_type put_call_mid;
30
31 Put ← Actor(0, PUT, put_cid, put_call_mid, put_call, put_ret,
32             put_EET_before, put_EET_after);
33
34 // to Shared Object Controller
35 broadcast chan method_req[NC];
36 chan method_grant[NC];
37 // from Shared Object Controller
38 urgent chan method_complete[NC];
39
40 Put_Port ← Port(put_cid, put_call_mid, put_call, put_ret,
41                 method_req, method_grant,
42                 mid_request, call_status,
43                 method_complete);
44
45 urgent chan get_call, get_ret;
46 client_type get_cid;
47 method_type get_call_mid;
48
49 Get ← Actor(1, GET, get_cid, get_call_mid, get_call, get_ret,
50             get_EET_before, get_EET_after);
51
52 Get_Port ← Port(get_cid, get_call_mid, get_call, get_ret,
53                 method_req, method_grant,
54                 mid_request, call_status,
55                 method_complete);
56
57 // to Arbiter
58 urgent broadcast chan arbitrate;
59 urgent chan so_grant, exec_method, done_method;
60 method_type scheduled_mid;
61
62 SO_Ctlr ← SO_Controller(method_req,
63                         arbitrate, so_grant, granted_cid,
64                         method_grant,
65                         exec_method, done_method,
66                         method_complete);
67
68 // Arbiter to Server
69 urgent chan call_so, ret_so;
70 // Arbiter to Guard Evaluator
71 urgent chan eval_guards, eval_guards_done;
72
73 SO_GE ← SO_Guard_Evaluator(eval_guards, eval_guards_done,
74                            mid_request, mid_request_guarded,
75                            num_elements,
76                            EET_eval);
77
78 SO_Arb ← SO_Arbiter(arbitrate,
79                     eval_guards, eval_guards_done,
80                     so_grant,
81                     mid_request, mid_request_guarded, call_status,
82                     exec_method, done_method,
83                     call_so, granted_cid, scheduled_mid, ret_so,
```

```
84                          EET_sched);
85
86  SO_Beh ← SO_Behavior(call_so, ret_so, scheduled_mid,
87                          EET_service, num_elements);
88
89  system Put, Put_Port,
90         Get, Get_Port,
91         SO_Ctlr, SO_GE, SO_Arb, SO_Beh;
```

Listing B.3: System definition

B.3 Virtual Target Architecture Layer TA example

```
1   // definition of timing annotation intervals
2   typedef int[0,1] EET_type;
3   const EET_type BCET ← 0; // lower bound: Best−Case Execution Time (BCET)
4   const EET_type WCET ← 1; // upper bound: Worst−Case Execution Time (WCET)
5
6   // number of client processes
7   const int NC ← 2;
8   typedef int[0,NC−1] client_type;
9
10  // definition of method ID types
11  typedef int[0,2] method_type;
12  const method_type NOP ← 0; // no OPeration (NOP) represents no service call
13  const method_type PUT ← 1; // represents the put service
14  const method_type GET ← 2; // represents the get service
15
16  // definition of status ID types
17  typedef int[0,2] status_type;
18  const status_type WAIT ← 0;
19  const status_type GRANTED ← 1;
20  const status_type COMPLETED ← 2;
21
22  // index type for Shared Object Scheduler
23  typedef int[−1,NC−1] index_type;
24  index_type granted_mid ← −1; //initially no client is granted
25
26  // definition of FIFO size
27  const int FIFO_SIZE ← 5;
28
29  // definition of method argument and return value switches
30  typedef int[0,1] method_argument_size_type;
31  const method_argument_size_type ARG ← 0;
32  const method_argument_size_type RET ← 1;
33
34  // definition of method agrument sizes
35  //
36  //|      | argument    | return      |
37  //|  ID  | size [bit]  | size [bit]  |
38  //|−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−|
39  //| NOP  | 0           | 0           |
40  //| PUT  | 128         | 0           |
41  //| GET  | 0           | 128         |
42  //|−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−|
43  int method_argument_size[3][2] ← { {0,0}, {128,0}, {0,128} };
44
45  // definition of RMI phase switches
46  typedef int[0,6] RMI_data_type;
47  const RMI_data_type NO_DATA ← 0;
48  const RMI_data_type REQUEST ← 1;
49  const RMI_data_type PARAMS ← 2;
50  const RMI_data_type RETURN ← 3;
51
52  // RMI data phases
53  //
54  //|  phase   | NOP | PUT | GET |
55  //|−−−−−−−−−−−−−−−−−−−−−−−−−−−−|
```

```
56  //| NO_DATA  | 0    | 0    | 0    |
57  //| REQUEST  | 0    | 1    | 2    |
58  //| PARAMS   | 0    | 3    | 4    |
59  //| RETURN   | 0    | 5    | 6    |
60  //|-----------------------------|
61  int RMI_data_phase[4][3] ← { {0,0,0}, {0,1,2}, {0,3,4}, {0,5,6} };
62
63  /* ******************************************************************* */
64
65  // Description of the fully synchronized bus, following the OSI terminology:
66  // The variable wire models the shared bus access at the PHYSICAL LAYER
67  int[0,6] wire ← 0;
68  // rnw ← read not write (defines whether a read or a write access on
69  //                       the "wire" is performed)
70  bool rnw ← true;
71  // The channel send models the sender synchronization with the Bus
72  urgent chan send;
73  // The broadcast channel models bus and receiver synchronization
74  urgent broadcast chan receive;
75
76  // A simple transmission takes D1 time units
77  const int D1 ← 2;
78  // A burst transaction takes D1+(D2*(burst_length-1)) time units
79  const int D2 ← 1;
80
81  // DATA LINK Layer
82  // Bus addresses range from 1 to P , the adress zero denotes broadcast
83  const int P ← 2; // number of bus addresses (connection points)
84  int[0,P] busaddr ← 0;
85  bool bus_burst ← false;
86  int bus_burst_length ← 0;
87
88  // MEDIUM ACCESS is through a transmitter and reciever local for each
        connection
89  // transmitter and reciever request parameters
90  int buffer;        // buffer
91  int[0,P] address; // address
92  bool burst;
93  int burst_length;
94
95  // connections for transmitter and reciever
96  // sendrequest sr, recieverequest rr, endrecieve er:
97  urgent chan sr[P], rr[P], er[P];
98  urgent chan srr[P];
99
100 // Arbitration is done through busreq and grant channels
101 urgent broadcast chan busreq;
102 urgent broadcast chan grant;
103 bool req[P];
```

Listing B.4: Global definitions

```
1   // Timing annotations, written as [BCET, WCET] intervals
2   int put_EET_before[2] ← {10, 15};
3   int put_EET_after[2]  ← {4, 5};
4
5   int get_EET_before[2] ← {10, 15};
6   int get_EET_after[2]  ← {4, 5};
7
8   int EET_eval[2] ← {1, 1};
9   int EET_sched[2] ← {1, 1};
10
11  int[0,FIFO_SIZE] num_elements ← 0;
12  // Shared Object service execution time [PUT, GET][BCET, WCET]
13  int EET_service[2][2] ← {{2, 5}, {2, 5}};
14
15  // RMI initiator execution times [client0, client1][BCET, WCET]
16  int rmi_port_init[2][2] ← { {1,1}, {1,1} };
17  int rmi_port_lookup[2][2] ← { {1,2}, {1,2} };
18  int rmi_port_serialize_base[2][2] ← { {2,3}, {2,3} };
19  int rmi_port_deserialize_base[2][2] ← { {1,2}, {1,2} };
```

```
20 int rmi_port_final[2][2] ← { {1,1}, {1,1} };
21
22 // Timing analysis:
23 // global clock
24 clock t;
25 // timing requirements for put and get clients
26 const int PUT_PERIOD ← 55;
27 const int GET_PERIOD ← 55;
28
29 method_type mid_request[NC];
30 method_type mid_request_guarded[NC];
31 method_type so_mid ← NOP;
32
33 status_type call_status[NC];
34
35 urgent chan put_call, put_ret;
36 client_type put_cid;
37 method_type put_call_mid;
38
39 Put ← Actor(0, PUT, put_cid, put_call_mid, put_call, put_ret,
40             put_EET_before, put_EET_after);
41
42 // to RMI Socket
43 broadcast chan method_req[NC];
44 urgent chan params_streamed;
45 urgent chan done_RMI;
46
47 Put_Port ← RMIPort(put_cid, put_call_mid, put_call, put_ret,
48                    method_req, params_streamed, done_RMI,
49                    mid_request, call_status,
50                    rmi_port_init,
51                    rmi_port_lookup,
52                    rmi_port_serialize_base,
53                    rmi_port_deserialize_base,
54                    rmi_port_final,
55                    1, 1, 32);
56
57 urgent chan get_call, get_ret;
58 client_type get_cid;
59 method_type get_call_mid;
60
61 Get ← Actor(1, GET, get_cid, get_call_mid, get_call, get_ret,
62             get_EET_before, get_EET_after);
63
64 Get_Port ← RMIPort(get_cid, get_call_mid, get_call, get_ret,
65                    method_req, params_streamed, done_RMI,
66                    mid_request, call_status,
67                    rmi_port_init,
68                    rmi_port_lookup,
69                    rmi_port_serialize_base,
70                    rmi_port_deserialize_base,
71                    rmi_port_final,
72                    2, 1, 32);
73
74 // to Arbiter
75 urgent broadcast chan arbitrate;
76 urgent chan so_grant;
77 urgent chan exec_method;
78 method_type scheduled_mid;
79 urgent chan done_method;
80
81 // to Receiver
82 int read_data;
83 int write_data;
84
85 SO_Ctrl ← SO_Controller(method_req,
86                           arbitrate, so_grant, params_streamed, exec_method,
       granted_mid,
87                           done_method, done_RMI,
88                           read_data, write_data);
89
```

```
90  // Arbiter to Shared
91  urgent chan call_so;
92  urgent chan ret_so;
93
94  // Arbiter to Guard Evaluator
95  urgent chan eval_guards;
96  urgent chan eval_guards_done;
97
98  SO_GE ← SO_Guard_Evaluator(eval_guards, eval_guards_done,
99                             mid_request, mid_request_guarded,
100                            num_elements,
101                            EET_eval);
102
103 SO_Arb ← SO_Arbiter(arbitrate,
104                     eval_guards, eval_guards_done,
105                     so_grant, exec_method,
106                     mid_request, mid_request_guarded, call_status,
107                     done_method,
108                     call_so, granted_mid, scheduled_mid, ret_so,
109                     EET_sched);
110
111 SO_Beh ← SO_Behavior(call_so, ret_so, scheduled_mid,
112                      EET_service, num_elements);
113
114 bus ← Bus(D1, D2);
115 bus_transmitter_0 ← Bus_Transmit(1);
116 bus_transmitter_1 ← Bus_Transmit(2);
117 bus_receiver_0 ← Bus_Receiver(1);
118 bus_receive_0 ← Bus_Receive(1, read_data, write_data);
119 bus_arbiter ← Bus_Arbiter();
120
121 system Put, Put_Port,
122        Get, Get_Port,
123        SO_Ctrl, SO_GE, SO_Arb, SO_Beh,
124        bus_transmitter_0, bus_transmitter_1,
125        bus_receiver_0, bus_receive_0,
126        bus, bus_arbiter;
```

Listing B.5: System definition

APPENDIX C

Pre-defined Shared Objects

When mapping Behavior Layer models to Application Layer models, all remaining Behavior Layer Channels of kind *Shared Variable* (see Listing C.1), *Piped Variable* (see Listing C.5), *Queue* (see Listing C.4), *Handshake* (see Listing C.2) and *Double Handshake* (see Listing C.3) are replaced by their Shared Object implementations listed below.

```
template<class T>
class Shared_Variable : public read_write_if<T> {
 public:
  Shared_Variable() : m_data() {}

  OSSS_GUARDED_METHOD_VOID(write, OSSS_PARAMS(1, T, item), true ) { m_data = item; }
  OSSS_GUARDED_METHOD(T, read, OSSS_PARAMS(0), true ) { return m_data; }

 protected:
  T m_data;
};
```

Listing C.1: Shared Variable as Shared Object

```
template<class T>
class Handshake : public send_receive_if<T> {
 public:
  Handshake() : m_data(), m_data_ready(false) {}

  OSSS_GUARDED_METHOD_VOID(send, OSSS_PARAMS(1, T, item), true ) {
    m_data = item;
    m_data_ready = true;
  }

  OSSS_GUARDED_METHOD(T, receive, OSSS_PARAMS(0), m_data_ready ) {
    m_data_ready = false;
    return m_data;
  }

 protected:
  T m_data;
  bool m_data_ready;
};
```

Listing C.2: Handshake Channel as Shared Object

```
 1  template<class T>
 2  class Double_Handshake_if : public send_receive_if<T> {
 3   public:
 4    virtual void wait_for_receive() = 0;
 5  };
 6
 7  template<class T>
 8  class Double_Handshake : public Double_Handshake_if<T> {
 9   public:
10    Double_Handshake() : m_data(), m_data_ready(false) {}
11
12    OSSS_GUARDED_METHOD_VOID(send, OSSS_PARAMS(1, T, item), !m_data_ready ) {
13      m_data = item;
14      m_data_ready = true;
15    }
16
17    OSSS_GUARDED_METHOD_VOID(wait_for_receive, OSSS_PARAMS(0), !m_data_ready) {}
18
19    OSSS_GUARDED_METHOD(T, receive, OSSS_PARAMS(0), m_data_ready ) {
20      m_data_ready = false;
21      return m_data;
22    }
23
24   protected:
25    T m_data;
26    bool m_data_ready;
27  };
```

Listing C.3: Double Handshake Channel as Shared Object

```
 1  template<class T, unsigned int Size>
 2  class Queue : public put_get_if<T> {
 3   public:
 4    Queue() : m_put_index(0),
 5              m_get_index(0),
 6              m_num_items(0) {}
 7
 8    OSSS_GUARDED_METHOD_VOID(put, OSSS_PARAMS(1, T, item), !OSSS_EXPORTED(is_full()) ) {
 9      m_buffer[m_put_index] = item;
10      increment_index(m_put_index);
11      m_num_items += 1;
12    }
13
14    OSSS_GUARDED_METHOD(T, get, OSSS_PARAMS(0), !OSSS_EXPORTED(is_empty()) ) {
15      T result = m_buffer[m_get_index];
16      increment_index(m_get_index);
17      m_num_items -= 1;
18      return result;
19    }
20
21    OSSS_GUARDED_METHOD(bool, is_empty, OSSS_PARAMS(0), true) {
22      return m_num_items == 0;
23    }
24
25    OSSS_GUARDED_METHOD(bool, is_full, OSSS_PARAMS(0), true) {
26      return m_num_items == Size;
27    }
28
29   protected:
30
31    void increment_index(unsigned int &index) {
32      if (index == (Size-1)) index = 0;
33      else index += 1;
34    }
35
36    T m_buffer[Size];
37    unsigned int m_put_index, m_get_index, m_num_items;
38  };
```

Listing C.4: Queue as Shared Object

```
template<class T>
class Piped_Variable_if : public read_write_if<T> {
public:
  virtual void start_write() = 0;
  virtual void end_write() = 0;
  virtual void start_read() = 0;
  virtual void end_read() = 0;
};

template<class T, unsigned int Depth>
class Piped_Variable : public Piped_Variable_if<T> {
 public:
  Piped_Variable() : m_data(),
                     m_in_write(false), m_in_read(false),
                     m_start_counter(0), m_access_counter(0) {}

  OSSS_GUARDED_METHOD_VOID(start_write, OSSS_PARAMS(0), !m_in_read) {
    m_in_write = true;
  }

  OSSS_GUARDED_METHOD_VOID(end_write, OSSS_PARAMS(0), m_in_write && !m_in_read) {
    m_access_counter++;
    if (m_start_counter < Depth) {
      step();
      m_access_counter = 0;
      m_start_counter++;
    }
    else if (m_access_counter == 2) {
      step();
      m_access_counter = 0;
    }
    m_in_write = false;
  }

  OSSS_GUARDED_METHOD_VOID(start_read, OSSS_PARAMS(0), !m_in_write) {
    m_in_read = true;
  }

  OSSS_GUARDED_METHOD_VOID(end_read, OSSS_PARAMS(0), m_in_read && !m_in_write) {
    m_access_counter++;
    if (m_access_counter == 2) {
      step();
      m_access_counter = 0;
    }
    m_in_read = false;
  }

  OSSS_GUARDED_METHOD_VOID(write, OSSS_PARAMS(1, T, item), m_in_write && !m_in_read) {
    m_data[0] = item;
  }

  OSSS_GUARDED_METHOD(T, read, OSSS_PARAMS(0), !m_in_write && m_in_read) {
    return m_data[Depth];
  }

 protected:
  void step() {
    for(int i=0; i<Depth; i++) {
      m_data[Depth-i] = m_data[Depth-(i+1)];
    }
  }

  T m_data[Depth+1];
  bool m_in_write, m_in_read;
  unsigned char m_start_counter, m_access_counter;
};
```

Listing C.5: Piped Variable Variable as Shared Object

Listing C.6 show the details of the `PAR` Shared Objects used to implement the fork-join semantics. The `PAR_master_if` provides services to initiate the `fork` and to wait for the completion of all forked processes (`join`). The `PAR_slave_if` provides services to each of the forked Actors. When each Actor is `ready` the execution of the `main` routine starts. After execution of its `main` routine, each Actor notifies completion (`done`) and waits for each of the other Actors to finish (`exit`).

```
1  class PAR_master_if : public virtual sc_interface {
2   public:
3    virtual void fork() = 0;
4    virtual void join() = 0;
5  };
6
7   class PAR_slave_if : public virtual sc_interface {
8   public:
9    virtual void ready(unsigned int id) = 0;
10   virtual void start() = 0;
11   virtual void done(unsigned int id) = 0;
12   virtual void exit() = 0;
13 };
14
15 template<unsigned int Size>
16 class PAR : public PAR_maste_if,
17             public PAR_slave_if {
18  public:
19   PAR() : m_init(false) {
20     for(int i=0; i<Size; i++) { m_ready[i] = false; }
21   }
22
23   OSSS_GUARDED_METHOD_VOID(fork, OSSS_PARAMS(1, unsigned int, id), !m_init ) {
24     for(int i=0; i<Size; i++) { m_ready[i] = false; }
25     m_init = true;
26   }
27
28   OSSS_GUARDED_METHOD_VOID(ready, OSSS_PARAMS(1, unsigned int, id), m_init ) {
29     m_ready[id] = true;
30   }
31
32   OSSS_GUARDED_METHOD_VOID(start, OSSS_PARAMS(0), all_ready() ) { }
33
34   OSSS_GUARDED_METHOD_VOID(done, OSSS_PARAMS(1, unsigned int, id), m_init ) {
35     m_ready[id] = false;
36   }
37
38   OSSS_GUARDED_METHOD_VOID(exit, OSSS_PARAMS(0), all_done() ) { }
39
40   OSSS_GUARDED_METHOD_VOID(join, OSSS_PARAMS(0), all_done() ) { m_init = false; }
41
42  protected:
43   bool all_ready() const {
44     for(int i=0; i<Size; i++) { if (!m_ready[i]) return false; }
45     return true;
46   }
47
48   bool all_done() const {
49     for(int i=0; i<Size; i++) { if (m_ready[i]) return false; }
50     return true;
51   }
52
53   bool m_init;
54   bool m_ready[Size];
55 };
```

Listing C.6: Shared Object for `PAR` Behavior fork-join semantics

Listing C.7 show the details of the `PIPE` Shared Objects used to implement the pipeline execution semantics.

The `PIPE_master_if` provides a service to set an upper bound of the execution of each pipeline stage. By default it is zero, resulting in an infinite execution of pipeline. The `fork` service starts the pipeline execution and, if the pipeline execution is bounded, `join` waits for the completion of the pipeline execution, similar to the `PAR` Shared Object.

The `PIPE_slave_if` provides services to each of the pipeline stage Actors. The `init` barrier is taken after the pipeline master has started the pipeline execution through `fork`. When each pipeline stage is `ready` the execution of the `main` routine starts only if the ramp-up phase for the corresponding stage has been completed. At the end of each pipeline stage's execution cycle the `done` call notifies completion of the cycle and checks if the bounded number of executions for

each stage has been reached. If this should be the case, the `exit` join-barrier is taken and the `init` barrier is entered again, waiting for the next pipeline activation. If the execution bound has not been reached or the pipeline execution is unbounded, the `exit` barrier is taken and the `ready` service is executed again.

```
1  class PIPE_master_if : public virtual sc_interface {
2   public:
3    virtual void set_limit(unsigned int limit) = 0;
4    virtual void fork() = 0;
5    virtual void join() = 0;
6  };
7
8  class PIPE_slave_if : public virtual sc_interface {
9   public:
10   virtual void init() = 0;
11   virtual void ready(unsigned int id) = 0;
12   virtual bool start(unsigned int id) = 0;
13   virtual bool done(unsigned int id) = 0;
14   virtual void exit() = 0;
15 };
16
17 template<unsigned int Stages>
18 class PIPE : public PIPE_master_if,
19              public PIPE_slave_if {
20  public:
21   PIPE() : m_init(false),
22            m_limit(0) {
23     for(int i=0; i<Stages; i++) {
24       m_ready[i] = false;
25       m_counter[i] = 0;
26     }
27   }
28
29   OSSS_GUARDED_METHOD_VOID(set_limit, OSSS_PARAMS(1, unsigned int, limit), true ) {
30     m_limit = limit;
31   }
32
33   OSSS_GUARDED_METHOD_VOID(fork, OSSS_PARAMS(1, unsigned int, limit), !m_init ) {
34     for(int i=0; i<Stages; i++) {
35       m_ready[i] = false;
36       m_counter[i] = 0;
37     }
38     m_init = true;
39   }
40
41   OSSS_GUARDED_METHOD_VOID(init, OSSS_PARAMS(0), m_init ) { }
42
43   OSSS_GUARDED_METHOD_VOID(ready, OSSS_PARAMS(1, unsigned int, id), m_init ) {
44     m_ready[id] = true;
45     if (m_limit != 0) { m_counter[id]++; }
46     else if (m_counter[id] < id) { m_counter[id]++; }
47   }
48
49   OSSS_GUARDED_METHOD(bool, start, OSSS_PARAMS(1, unsigned int, id), all_ready() ) {
50     return (m_counter[id] >= id);
51   }
52
53   OSSS_GUARDED_METHOD(bool, done, OSSS_PARAMS(1, unsigned int, id), m_init ) {
54     if (m_limit != 0) {
55       m_ready[id] = false;
56       return (m_counter[id] >= m_limit);
57     }
58     else {
59       m_ready[id] = false;
60       return false;
61     }
62   }
63
64   OSSS_GUARDED_METHOD_VOID(exit, OSSS_PARAMS(0), all_done() ) { }
65   OSSS_GUARDED_METHOD_VOID(join, OSSS_PARAMS(0), limit_reached() ) {
66     m_init = false;
67   }
68
69  protected:
70   bool all_ready() const {
71     for(int i=0; i<Stages; i++) { if (!m_ready[i]) return false; }
72     return true;
73   }
74
75   bool all_done() const {
76     for(int i=0; i<Stages; i++) { if (m_ready[i]) return false; }
77     return true;
78   }
```

```
79
80    bool limit_reached() const {
81      if (limit == 0) return false;
82      for(int i=0; i<Stages; i++) { if (m_ready[i] && m_counter[i] < m_limit) return false; }
83      return true;
84    }
85
86    bool m_init;
87    unsigned int m_limit;
88    bool m_ready[Stages];
89    unsigned int m_counter[Stages];
90  };
```

Listing C.7: Shared Object for `PIPE` Behavior semantics

APPENDIX D

I^2C Protocol OSSS Channel Implementation

D.1 Introduction

The I^2C (**I**nter-**I**ntegrated-**C**ircuit-Bus) Protocol is primarily used for the communication between integrated circuits as the name already implies. The complete specification can be found in [164]. This simplified I^2C bus protocol is a single master configuration and has been implemented based on [153] and demonstrates the full support of all Objective VHDL+ communication modeling capabilities by OSSS Channels.

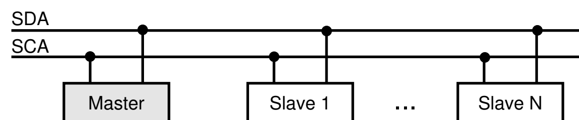


Figure D.1: Organization of the I^2C bus

The physical implementation of the I^2C bus is a bidirectional two wire bus. The SDA (**S**hift **DA**ta) line is used for the transportation of the data. The SCL (**S**hift **CL**ock) line is used for the synchronization of the data. In our modeling example exactly one bus master has to be connected to bus. It controls the data transfer on the bus. Furthermore, multiple slaves can be attached to the bus, but in this modeling experiment the number of slaves has been limited to two. The bus master initiates each communication on the bus and drives the SCL signal, which determines the bus clock. The slaves can send or receive data as requested by the bus master. While the SCL signal is only driven by the single master the SDA signal is either driven by the master or the slave depending on the direction of the data (the writer drives the SDA line). When the SDA line is driven by multiple writes the conflict is solved by a "wired and" function.

MSB							LSB
A_6	A_5	A_4	A_5	A_2	A_1	A_0	R/W

Table D.1: I^2C address format

Each slave module has a unique address which consists of eight bits as shown in Table D.1. A_6 down to A_0 are the address bits of each slave. The R/W bit is not part of the physical address but it is used by the master to indicate whether a read or a write transfer is initiated.

D.2 The I^{2}C Bus Protocol

The data transfer on the SDA line is serial (single signal) and synchronous to the SCL signal. When no communication takes place, both signals SDA and SCL are high. The master initiates a transfer with a *start condition* which wakes up all the attached slaves. A *start condition* is characterized by a falling edge of the SDA signal while the SCL signal remains high. The end of a communication is notified by the master performing the *stop condition*. A *stop condition* is characterized by a rising edge of the SDA signal while the SCL signal remains high. Figure D.2 illustrates the start and the stop conditions in a waveform.

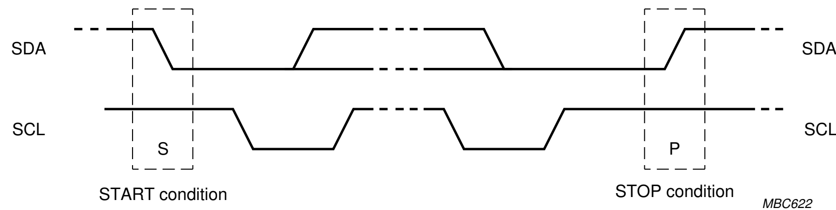


Figure D.2: I^{2}C start and stop conditions [164]

D.2.1 Data Transfer from Master to Slave

Table D.2a shows the data transfer from the master to a slave. After setting the start condition, the master initiates a write request to one of the slaves by transferring his address on the SDA line. It starts with the MSB, while the last bit indicates a read or a write transfer (setting the R/W bit to zero indicates a write transfer). Since all slaves are woken up by the master's start condition, each slave compares the address with its own. If it matches, the slave acknowledges the reception of the address by pulling down the SDL line synchronous to with the next clock event on the SCL line. Now the master sequentially clocks a data byte on the SDA line. The slave acknowledges the reception of the data byte in the same manner as it acknowledged the address reception by pulling down the SDA signal. The transmission finishes when either the slave does not accept the receipt of delivery (acknowledge = 1) and/or the master generates the stop condition.

S	7 bit slave address	R/W=0	A	8 bit data	A	...	A	8 bit data	A/$\overline{\text{A}}$	P

(a) data transfer from master to slave

S	7 bit slave address	R/W=1	A	8 bit data	A	...	A	8 bit data	$\overline{\text{A}}$	P

(b) data transfer from slave to master

S	start condition	A	acknowledge	$\overline{\text{A}}$	not acknowledge	P	stop condition

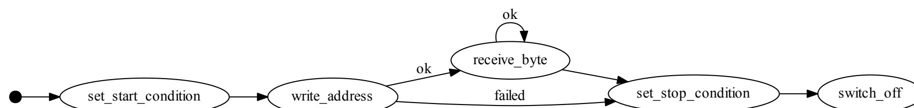
Table D.2: I^{2}C data transfer

D.2.2 Data Transfer from Slave to Master

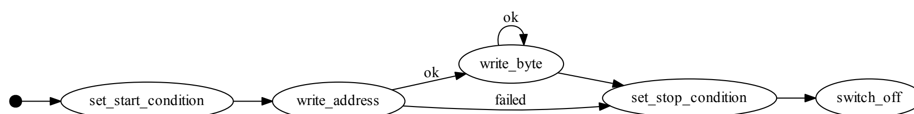
Table D.2b shows the data transfer from a slave to the master. After setting the start condition the master initiates a read request to one of the slaves by transferring his address on the SDA line. It starts with the MSB, while the last bit indicates a read or a read transfer (setting the R/W bit to one indicates a read transfer). Since all slaves are woken up by the master's start condition, each slave compares the address with its own. If it matches, the slave acknowledges the reception of the address by pulling down the SDL line synchronous to with the next clock event on the SCL line. Now the slave sequentially clocks a data byte on the SDA line synchronous to

the SCL signal. The master acknowledges (acknowledge = 0) the reception of the data byte on by pulling down the SDA signal. The transmission finishes when the master refuses the acknowledgment (acknowledge = 1) followed by the generation of the stop condition.

Figure D.3 shows the master protocol state machines and Figure D.4 shows the slave protocol state machine which have to be implemented in the master/slave transactors of the OSSS Channel.



(a) `master_receiving_byte`



(b) `master_send_byte`

Figure D.3: Protocol state machines for the bus master

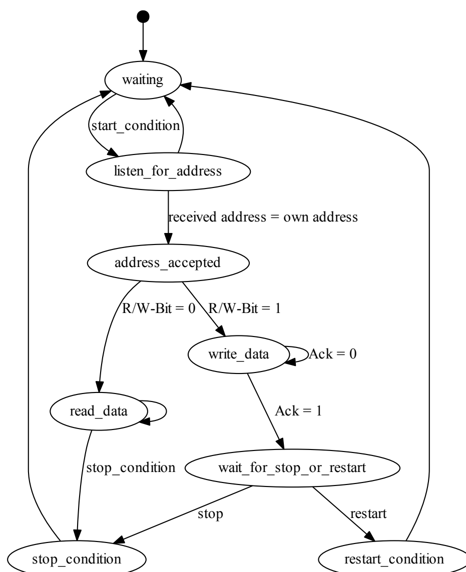


Figure D.4: Protocol state machine for the slave (`slave_action_byte`)

D.3 OSSS Channel implementation

```
typedef sc_uint<7> Address;
typedef sc_uint<8> CompleteAddress;
typedef sc_uint<8> SendData;
typedef sc_uint<8> ReceiveData;

class master_method_if : public osss_method_if {
public:
  virtual void master_send_byte(Address addr, SendData sendData) = 0;
  virtual void master_receive_byte(Address addr, ReceiveData &recData) = 0;
};

class slave_method_if : public osss_method_if {
public:
  virtual void slave_action_byte(Address &addr, SendData &sendData, ReceiveData &recData) = 0;
};
```

Listing D.1: I^{2}C transactor method interface

```
class master_signal_if : public osss_signal_if {
public:
  sc_inout<bool> shiftData;
  sc_out<bool> shiftClock;

  OSSS_GENERATE {
    osss_connect(osss_reg_port(shiftData), osss_shared_signal("shiftData"));
    osss_connect(osss_reg_port(shiftClock), osss_shared_signal("shiftClock"));
  }
};

class slave_signal_if : public osss_signal_if {
public:
  sc_inout<bool> shiftData;
  sc_in<bool> shiftClock;

  OSSS_GENERATE {
    osss_connect(osss_shared_signal("shiftData"), osss_reg_port(shiftData));
    osss_connect(osss_shared_signal("shiftClock"), osss_reg_port(shiftClock));
  }
};
```

Listing D.2: I^{2}C transactor signal interface

```
#define HIGH true
#define LOW false

enum State { waiting, listen_for_address, address_accepted, read_data, write_data,
             wait_for_stop_or_restart, stop_condition, restart_condition };

class Channel : public osss_basic_channel {
 public:
  class Transactor_master: public master_signal_if, public master_method_if {
     ...
  };

  class Transactor_slave: public slave_signal_if, public slave_method_if {
     ...
  };
};

OSSS_REGISTER_TRANSACTOR(Channel::Transactor_master, master_method_if)
OSSS_REGISTER_TRANSACTOR(Channel::Transactor_slave,  slave_method_if)
```

Listing D.3: I^{2}C channel with transactor implementations

```
1  class Transactor_master: public master_signal_if, public master_method_if {
2    public:
3
4      virtual void reset() {
5        shiftData.write(0);
6        shiftClock.write(0);
7      }
8
9      // intentional private methods for master
10     void set_start_condition() {
11       shiftData.write(HIGH);
12       shiftClock.write(HIGH);
13       wait();
14       shiftData.write(LOW);
15       wait();
16       shiftClock.write(LOW);
17       wait();
18     }
19
20     void write_address(CompleteAddress &completeAddress, bool &sendStatus) {
21       for (int i=0; i<8; i++) {
22         wait();
23         shiftData.write(completeAddress[i]);
24         wait();
25         shiftClock.write(HIGH);
26         wait(2);
27         shiftClock.write(LOW);
28         wait();
29       }
30       //wait for acknowledge
31       shiftData.write(HIGH);
32       wait();
33       shiftClock.write(HIGH);
34       wait();
35       if ( shiftData.read() == 0) {
36         sendStatus = true; //sending ok
37         wait();
38         shiftClock.write(LOW);
39       }
40       else {
41         sendStatus = false;
42         wait();
43         shiftClock.write(LOW);
44       }
45     }
46
47     void write_byte(SendData &sendData, bool &sendStatus) {
48       for(int i=0; i<8; i++) {
49         wait();
50         shiftData.write(sendData[i]);
51         wait();
52         shiftClock.write(HIGH);
53         wait(2);
54         shiftClock.write(LOW);
55         wait();
56       }
57       // wait for acknowledge
58       shiftData.write(HIGH);
59       wait();
60       shiftClock.write(HIGH);
61       wait();
62       if ( shiftData.read() == 0) {
63         sendStatus = true; //sending ok
64         wait();
65         shiftClock.write(LOW);
66       }
67       else {
68         sendStatus = false;
69         wait();
70         shiftClock.write(LOW);
71       }
72     }
73
74     void receive_byte(ReceiveData &receiveData) {
75       for (int i=0; i<8; i++) {
76         wait();
77         shiftClock.write(HIGH);
78         wait();
79         receiveData[i] = shiftData.read();
80         wait();
81         shiftClock.write(LOW);
82         wait();
83       }
84       // set acknowledge
```

```
85        shiftData.write(HIGH);
86        wait();
87        shiftClock.write(HIGH);
88        wait(2);
89        shiftClock.write(LOW);
90        wait();
91      }
92
93      void set_stop_condition() {
94        wait();
95        shiftData.write(LOW);
96        wait();
97        shiftClock.write(HIGH);
98        wait();
99        shiftData.write(HIGH);
100       wait();
101       wait();
102     }
103
104     void set_rec_stop_condition() {
105       shiftData.write(LOW);
106       wait();
107       shiftClock.write(HIGH);
108       wait();
109       shiftData.write(HIGH);
110       wait();
111     }
112
113     void set_restart_condition() {
114       wait();
115       shiftClock.write(HIGH);
116       shiftData.write(HIGH);
117       wait();
118       shiftData.write(LOW);
119       wait();
120       shiftClock.write(LOW);
121       wait();
122     }
123
124     void switch_off() {
125       shiftClock.write(HIGH);
126       shiftData.write(HIGH);
127       wait();
128     }
129
130     // public methods for master
131     virtual void master_send_byte(Address addr, SendData sendData) {
132       sc_uint< 8 > completeAddr;
133       bool sendStatus = false;  // false =  error, true = ok
134       completeAddr.range(6,0) = addr;
135       completeAddr[7] = 0;
136       set_start_condition();
137       write_address(completeAddr, sendStatus);
138       write_byte(sendData, sendStatus);
139       set_stop_condition();
140       switch_off();
141     }
142
143     virtual void master_receive_byte(Address addr, ReceiveData &recData) {
144       CompleteAddress completeAddr;
145       bool sendStatus = false;  // false =  error, true = ok
146       completeAddr.range(6,0) = addr;
147       completeAddr[7] = 1;
148       set_start_condition();
149       write_address(completeAddr, sendStatus);
150       receive_byte(recData);
151       set_stop_condition();
152       switch_off();
153     }
154   };
```

Listing D.4: I²C master transactor implementations

```
1  class Transactor_slave: public slave_signal_if, public slave_method_if {
2    public:
3      State current_state;
4      bool sda_old, sda_new, scl_old, scl_new;
5      CompleteAddress completeAddress;
6      bool rwMode;
7      bool not_stop_loop;
8      bool reading_ok;
9      ReceiveData data;
10
11     virtual void reset() {}
12
13     virtual void slave_action_byte(Address &addr, SendData &sendData, ReceiveData &recData) {
14       bool finished = false;
15       current_state = waiting;
16
17       while (!finished) {
18         switch ( current_state ) {
19
20           case waiting:
21             // waiting for start_condition
22             // if start condition listen for address
23             sda_old = shiftData.read();
24             scl_old = shiftClock.read();
25             wait();
26             sda_new = shiftData.read();
27             scl_new = shiftClock.read();
28             if ( ((sda_old == 1) && (sda_new == 0)) &&
29                  ((scl_old == 1) && (scl_new == 1)) ) { current_state = listen_for_address; }
30             else { current_state = waiting; }
31             break;
32
33           case listen_for_address:
34             // if address equals own address -> address_accepted()
35             // otherwise -> waiting()
36             // listen as well for the R/W signal
37             for (int i=0; i<8; i++) {
38               while(shiftClock.read() != 0) { wait(); }
39               while(shiftClock.read() != 1) { wait(); }
40               completeAddress[i] = shiftData.read();
41             }
42             rwMode = completeAddress[7];
43             if (completeAddress.range(6,0) == addr) { current_state = address_accepted; }
44             else { current_state = waiting; }
45             break;
46
47          case address_accepted:
48            // analyse the R/W bit -> read_data or write_data
49            // confirm received address
50            while(shiftClock.read() != 0) { wait(); }
51            wait();
52            shiftData.write(LOW); //acknowledge
53            wait();
54            while(shiftClock.read() != 1) { wait(); }
55            while(shiftClock.read() == 1) { wait(); }
56            // data from master to slave
57            if (rwMode == 0) { current_state = read_data; }
58            // data from slave to master
59            else { current_state = write_data; }
60            break;
61
62          case read_data:
63            reading_ok = false;
64            not_stop_loop = true;
65            for(int i=0; i<8; i++) {
66              while(shiftClock.read() != 0) { wait(); }
67              while(shiftClock.read() != 1) { wait(); }
68              while( (shiftClock.read()==1) && not_stop_loop ) {
69                sda_old = shiftData.read();
70                scl_old = shiftClock.read();
71                wait();
72                sda_new = shiftData.read();
73                scl_new = shiftClock.read();
74                if ( ((sda_old == 0) && (sda_new == 1)) &&
75                     ((scl_old == 1) && (scl_new == 1)) ) {
76                  not_stop_loop = false;
77                  current_state = stop_condition;
78                }
79              }
80              if (not_stop_loop == false) {
81                current_state = stop_condition;
82                break;
83              }
84              else { data[i] = sda_old; }
```

```
85          }
86          if (current_state != stop_condition) {
87            //confirm received data
88            while(shiftClock.read() != 0) { wait(); }
89            shiftData.write(LOW);
90            wait();
91            reading_ok = true;
92            while(shiftClock.read() != 1) { wait(); }
93            while(shiftClock.read() == 1) { wait(); }
94            if (reading_ok) { recData = data; }
95          }
96          break;
97
98        case write_data:
99          for (int i=0; i<8; i++) {
100           while(shiftClock.read() != 0) { wait(); }
101           wait();
102           shiftData.write( sendData[i] );
103           wait();
104           while(shiftClock.read() != 1) { wait(); }
105         }
106         while(shiftClock.read() != 0) { wait(); }
107         while(shiftClock.read() != 1) { wait(); }
108         wait();
109         if (shiftData.read() == 0) { current_state = write_data; }
110         else { current_state = wait_for_stop_or_restart; }
111         break;
112
113       case wait_for_stop_or_restart:
114         not_stop_loop = true;
115         while(shiftClock.read() != 0) { wait(); }
116         while(shiftClock.read() != 1) { wait(); }
117         while( (shiftClock.read() == 1) && not_stop_loop ) {
118           sda_old = shiftData.read();
119           scl_old = shiftClock.read();
120           wait();
121           sda_new = shiftData.read();
122           scl_new = shiftClock.read();
123           if ( ((scl_old == 1) && (scl_new == 1)) &&
124                ((sda_old == 0) && (sda_new == 1)) ) {
125             current_state = stop_condition;
126             not_stop_loop = false;
127           }
128           else {
129             if ( ((scl_old == 1) && (scl_new == 1)) &&
130                  ((sda_old == 1) && (sda_new == 0)) ) {
131               current_state = restart_condition;
132               not_stop_loop = false;
133             }
134           }
135         }
136         break;
137
138       case stop_condition:
139         current_state = waiting;
140         finished = true;
141         break;
142
143       case restart_condition:
144         current_state = listen_for_address;
145         break;
146     }
147   }
148 }
149 };
```

Listing D.5: I²C slave transactor implementations

APPENDIX E

Supported Target Platforms

This chapter gives and introduction to the supported FPGA-based target platforms. For more details please follow the references to the appropriate technical manuals and descriptions.

E.1 Supported FPGAs

Xilinx is the biggest FPGA manufacturer and offers a variety of Field Programmable Gate Arrays (from low-cost to high-density). Xilinx also offers the possibility to migrate from an FPGA prototype design to an ASIC like device, reducing the costs per chip, called EasyPath FPGAs. Together with their FPGAs Xilinx provides the ISE design software which in conjunction with the Xilinx Platform Studio can be used for the integration of various IP components including embedded processors, DSP blocks, interfaces & peripherals, and communication IPs. Xilinx offers its own embedded processors called PicoBlaze and MicroBlaze which can be connected with all offered IP components with IBMs CoreConnect interconnection technology. Both PicoBlaze and MicroBlaze are soft core processors which can be fully customized by the platform designer.

E.1.1 Virtex-4

Virtex-4 devices are user-programmable gate arrays with various configurable elements and embedded cores optimized for high-density and high-performance system designs. Virtex-4 devices implement the following functionality [234, 91]:

- I/O blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by programmable I/O blocks (IOBs).
- Configurable Logic Blocks (CLBs), the basic logic elements for Xilinx FPGAs, provide combinatorial and synchronous logic as well as distributed memory and SRL16 shift register capability.
- Block RAM modules provide flexible 18Kbit true dual-port RAM, which is cascadable to form larger memory blocks. In addition, Virtex-4 block RAMs contain optional programmable FIFO logic for increased device utilization.
- Cascadable embedded XtremeDSP slices with 18-bit x 18-bit dedicated multipliers, integrated Adder, and 48-bit accumulator.
- Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication/division, and coarse-/fine-grained clock phase shifting.

- The general routing matrix (GRM) provides an array of routing switches between each component. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs. All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

The Virtex-4 family is Xilinx' high-performance FPGA series and is manufactured with 90nm process technology. It offers three platforms tailored to the requirements of different application domains. The LX platform is intended for high-performance logic and has the highest logical cells to feature ratio and the highest I/O to feature ratio within the Virtex-4 family. The SX platform is intended for high-performance digital signal processing and therefore has the highest DSP block to feature ratio and the highest memory to feature ratio. The FX platform is intended for embedded processing and high speed serial connectivity. It additionally includes IBM PowerPC processor(s), Ethernet MAC blocks and RocketIO multi gigabit serial transceiver(s).

E.1.2 Virtex-II Pro

The Virtex-II Pro and families contain plat-form FPGAs for designs that are based on IP cores and customized modules. The family incorporates multi-gigabit transceivers and PowerPC CPU blocks in Virtex-II Pro Series FPGA architecture. It empowers complete solutions for telecommunication, wireless, networking, video, and DSP applications.

Virtex-II Pro devices are user-programmable gate arrays with various configurable elements and embedded blocks optimized for high-density and high-performance system designs. Virtex-II Pro devices implement the following functionality [20]:

- Embedded high-speed serial transceivers enable data bit rate up to 3.125 Gb/s per channel (RocketIO) or 6.25 Gb/s (RocketIO X).

- Embedded IBM PowerPC 405 RISC processor blocks provide performance up to 400 MHz.

- SelectIO-Ultra blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by the programmable IOBs.

- Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.

- Block SelectRAM+ memory modules provide large 18 Kb storage elements of True Dual-Port RAM.

- Embedded multiplier blocks are 18-bit x 18-bit dedicated multipliers.

- Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, and coarse- and fine-grained clock phase shifting.

- The general routing matrix (GRM) is an array of rout-ing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and supports high-speed designs.

All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

E.2 Supported Prototyping and Development Boards

In order to evaluate hardware/software designs using Xilinx FPGAs different development and prototyping boards are available through Xilinx and their partner companies. The different prototyping and development boards mainly differ in the peripheral components beside the FPGA(s) mounted on the PCB. Each prototyping board is equipped with at least one FPGA, onboard oscillator, flash memory, external memory and a JTAG interface for configuration/programming/debugging of the FPGA. Optionally other I/O hardware components with their interfaces are mounted on the boards (i.e. switches, LEDs, push buttons, LCD displays, RS232 interfaces, Ethernet, USB PHYs or A/D and D/A converters).

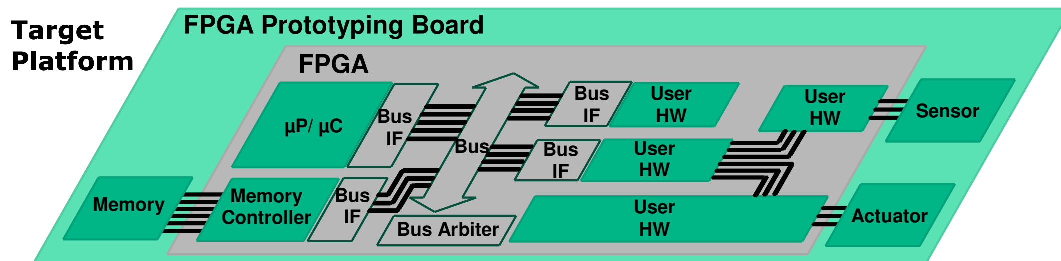


Figure E.1: Target platform showing an FPGA integrated in a prototyping board

Figure E.1 depicts the target platform which is situated at the bottom of Figure 6.1. In this case all architectural elements of the Virtual Target Architecture will be implemented on an FPGA. Usually different peripherals, connectors and interfaces connected to the FPGA are available on a prototyping board. In Figure E.1 these external components are labeled as Memory, Sensor and Actuator.

To support the evaluation of the methodology we have chosen two FPGA prototyping boards. The Xilinx ML401 and the Xilinx University Program Virtex-II Pro Development System (XUPII). Both boards have been chosen because of the supported interfaces, the flexibility of the FPGA, RTL synthesis tool support, and the portfolio of ready-to-use IP components. Both boards support important industry-standard peripherals, connectors and interfaces. The FPGAs, a Virtex-4 on the ML401, and a Virtex-II Pro on the XUPII, support implementation of custom hardware including soft core processors, and platform IP components. The Virtex-II Pro contains two high-performance Power PC hard macro processors.

E.2.1 The Xilinx ML401 Evaluation Platform

The Xilinx ML401 evaluation platform carries a Virtex-4 LX25 FPGA together with different industry-standard peripherals, connectors, and interfaces. This makes the ML401 evaluation platform to an ideal evaluation environment for a wide range of applications. Figure E.2 shows the Xilinx Virtex-4 ML401 evaluation platform block diagram.

The key features of the Xilinx ML401 evaluation platform are [85]:

- Virtex-4 FPGA (XC4VLX25-FF668-10)
- 64-MB DDR SDRAM, 32-bit interface running up to 266-MHz data rate
- One differential clock input pair and differential clock output pair with SMA connectors
- One 100-MHz clock oscillator (socketed) plus one extra open 3.3V clock oscillator socket
- General purpose DIP switches, LEDs, and push buttons
- Expansion header with 32 single-ended I/O, 16 LVDS capable differential pairs, 14 spare I/Os shared with buttons and LEDs, power, JTAG chain expansion capability, and IIC bus expansion

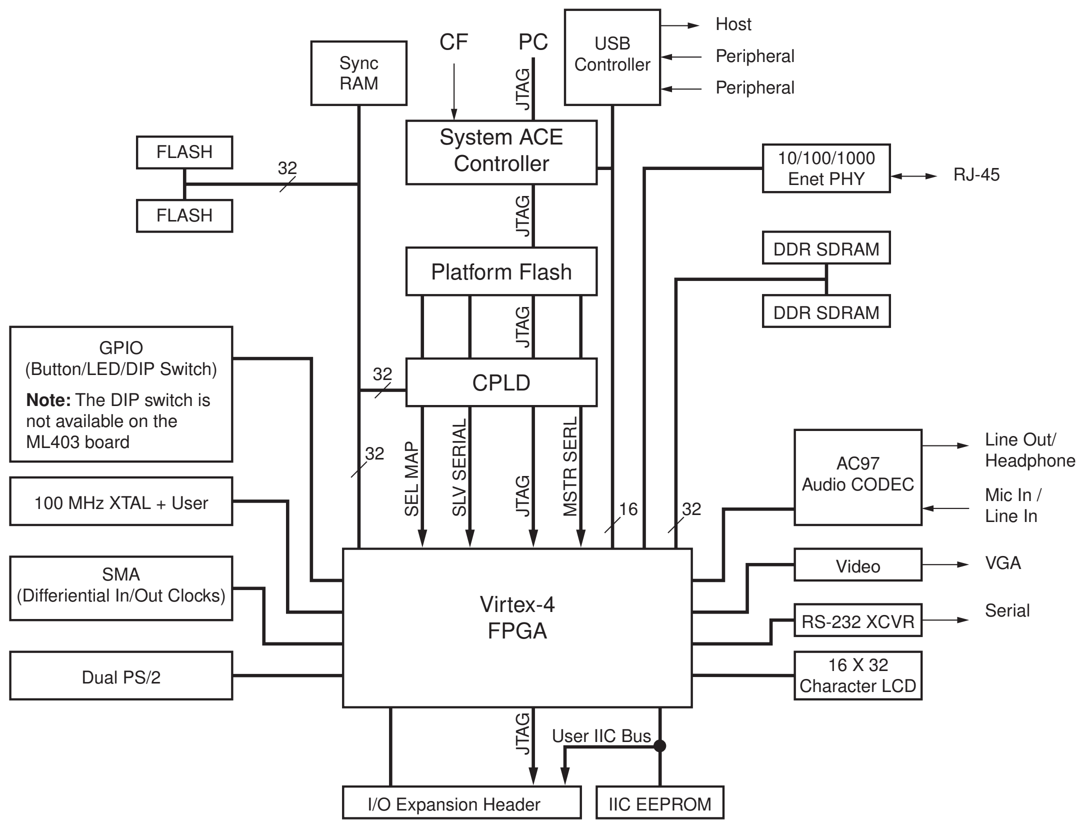


Figure E.2: Xilinx Virtex-4 ML401 evaluation platform block diagram [85]

- Stereo AC97 audio codec with line-in, line-out, 50-mW headphone, and microphone-in (mono) jacks
- RS-232 serial port
- 16-character x 2-line LCD display
- One 4-Kb IIC EEPROM
- VGA output: 50 MHz / 24-bit video DAC
- PS/2 mouse and keyboard connectors
- System ACE™ CompactFlash configuration controller with Type I/II CompactFlash connector
- ZBT synchronous SRAM: 9 MB on 32-bit data bus with four parity bits
- Intel StrataFlash (or compatible) linear flash chips (8 MB)
- 10/100/1000 tri-speed Ethernet PHY transceiver
- USB interface chip (Cypress CY7C67300) with host and peripheral ports
- Xilinx XC95144XL CPLD to allow linear flash chips to be used for FPGA configuration
- Xilinx XCF32P Platform Flash configuration storage device

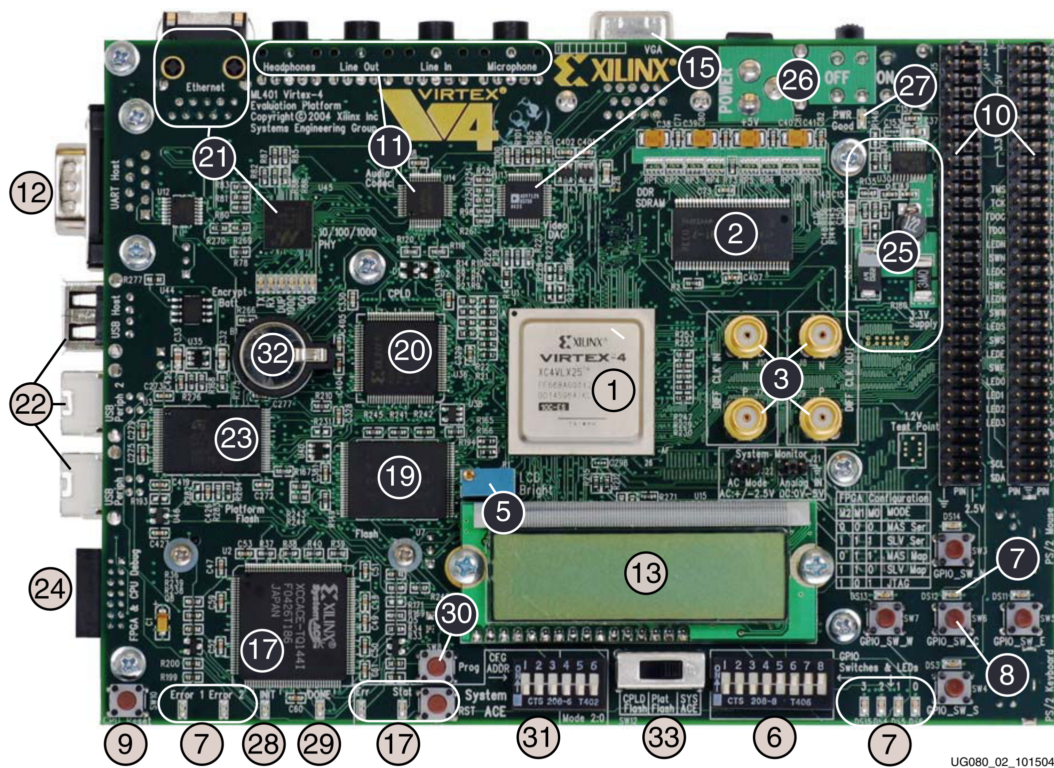


Figure E.3: Detailed description of Virtex-4 ML401 evaluation platform components (front) [85]

1. Virtex-4 FPGA
2. DDR SDRAM
3. Differential Clock Input And Output With SMA Connectors
4. Oscillator Sockets
5. LCD Brightness and Contrast Adjustment
6. DIP Switches (Active-High)
7. User and Error LEDs (Active-High)
8. User Push Buttons (Active-High)
9. CPU Reset Button (Active-Low)
10. Expansion Headers
11. Stereo AC97 Audio Codec
12. RS-232 Serial Port
13. 16-Character x 2-Line LCD
14. IIC Bus with 4-Kb EEPROM
15. VGA Output
16. PS/2 Mouse and Keyboard Ports
17. System ACE and CompactFlash Connector
18. ZBT Synchronous SRAM
19. Linear Flash Chips
20. Xilinx XC95144XL CPLD
21. 10/100/1000 Tri-Speed Ethernet PHY
22. USB Controller with Host and Peripheral Ports
23. Xilinx XCF32P Platform Flash Configuration Storage Device
24. JTAG Configuration Port
25. Onboard Power Supplies
26. AC Adapter and Input Power Switch/-Jack
27. Power Indicator LED
28. INIT LED
29. DONE LED
30. Program Switch
31. Configuration Address and Mode DIP Switches
32. Encryption Key Battery
33. Configuration Source Selector Switch

In the following we will briefly present the Virtex-4 FPGA family and will summarize the main features of the Virtex-4 LX25 mounted on the ML401 evaluation platform described above. The main features of the utilized Virtex-4 LX25 FPGA are [91]:

- 24,192 Logic Cells (LCs)

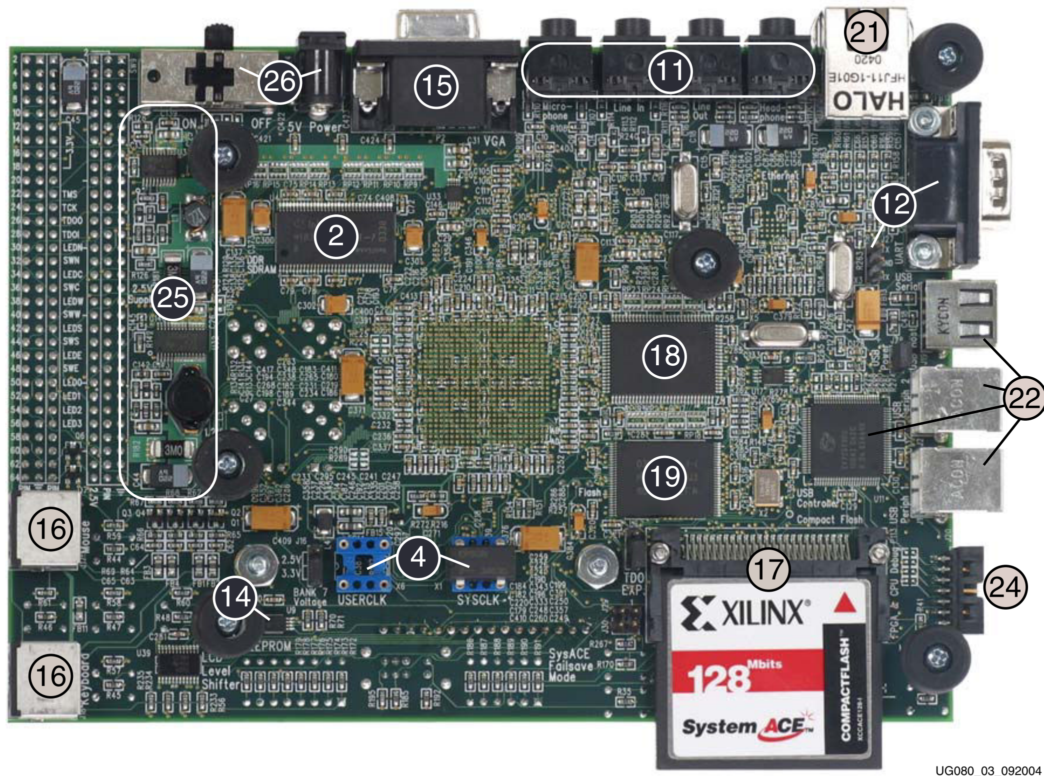


Figure E.4: Detailed description of Virtex-4 ML401 evaluation platform components (back) [85]

- 72 Block RAM/FIFO w/ECC (18 Kbits each = 1,296 Kbits of total block BRAM)
- 48 Extreme DSP Slices
- On- and off-chip system timing management using 8 DCMs (Digital Clock Managers) and 4 PMCD (Phase-Matched Clock Dividers)
- Single-ended electrical standard support for LVTTL, LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V), PCI (33 and 66 MHz), PCI-X, GTL and GTL+, HSTL 1.5V and 1.8V (Class I, II, III, and IV), and SSTL 2.5V and 1.8V (Class I and II).
- Differential electrical standard support for 840 LVDS, Extended LVDS (2.5V), Bus LVDS, ULVDS, LVPECL 2.5V, and HyperTransport (LDT). All I/Os can be configured as differential I/O without any placement restriction for flexibility.
- Built-in double-data rate input and output registers enable implementation of DDR and QDR interfaces.
- Seventeen I/O banks support electrical standards spanning across multiple voltages with independent reference voltages.
- Supports PicoBlaze and MicroBlaze embedded soft processor cores.

E.2.2 The Xilinx University Program Virtex-II Pro Development Board

The Xilinx University Program Virtex-II Pro Development Board carries a Virtex-II Pro XC2VP30 FPGA together with different industry-standard peripherals, connectors, and interfaces. Figure E.5 shows the development board block diagram.

The key features of the Xilinx University Program Virtex-II Pro Development Board are [34]:

- Virtex™-II Pro FPGA with PowerPC™ 405 cores

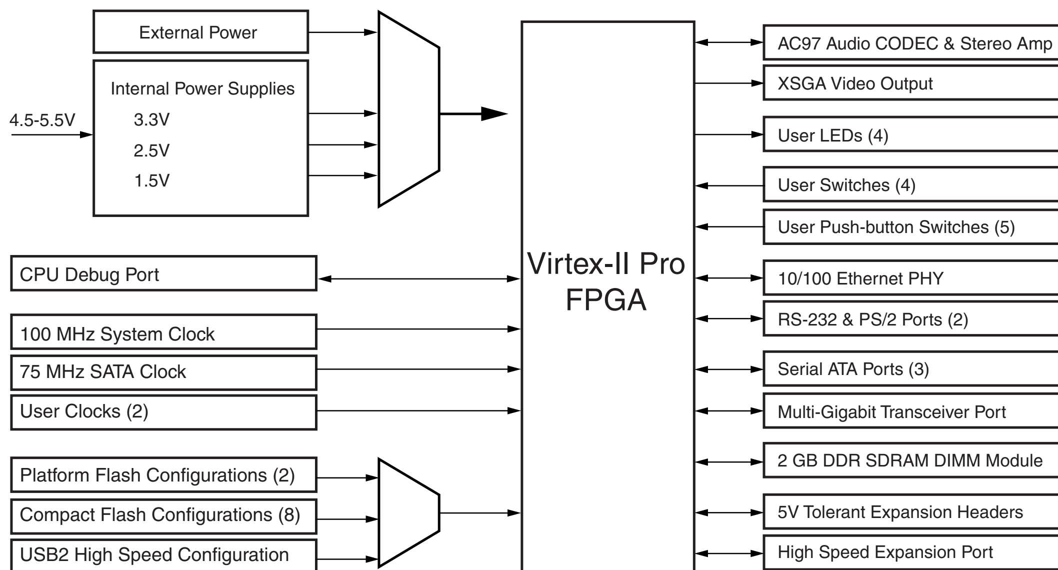


Figure E.5: XUP Virtex-II Pro Development System Block Diagram [34]

- Up to 2 GB of Double Data Rate (DDR) SDRAM
- System ACE™ controller and Type II CompactFlash™ connector for FPGA configuration and data storage
- Embedded Platform Cable USB configuration port
- High-speed SelectMAP FPGA configuration from Platform Flash In-System Programmable Configuration PROM
- Support for "Golden" and "User" FPGA configuration bitstreams
- On-board 10/100 Ethernet PHY device
- Silicon Serial Number for unique board identification
- RS-232 DB9 serial port
- Two PS-2 serial ports
- Four LEDs connected to Virtex-II Pro I/O pins
- Four switches connected to Virtex-II Pro I/O pins
- Five push buttons connected to Virtex-II Pro I/O pins
- Six expansion connectors joined to 80 Virtex-II Pro I/O pins with over-voltage protection
- High-speed expansion connector joined to 40Virtex-II Pro I/O pins that can be used differentially or single ended
- AC-97 audio CODEC with audio amplifier and speaker/headphone output and line level output
- Microphone and line level audio input
- On-board XSGA output, up to 1200 x 1600 at 70 Hz refresh
- Three Serial ATA ports, two Host ports and one Target port
- Off-board expansion MGT link, with user-supplied clock

- 100 MHz system clock, 75 MHz SATA clock
- Provision for user-supplied clock
- On-board power supplies
- Power-on reset circuitry
- PowerPC 405 reset circuitry

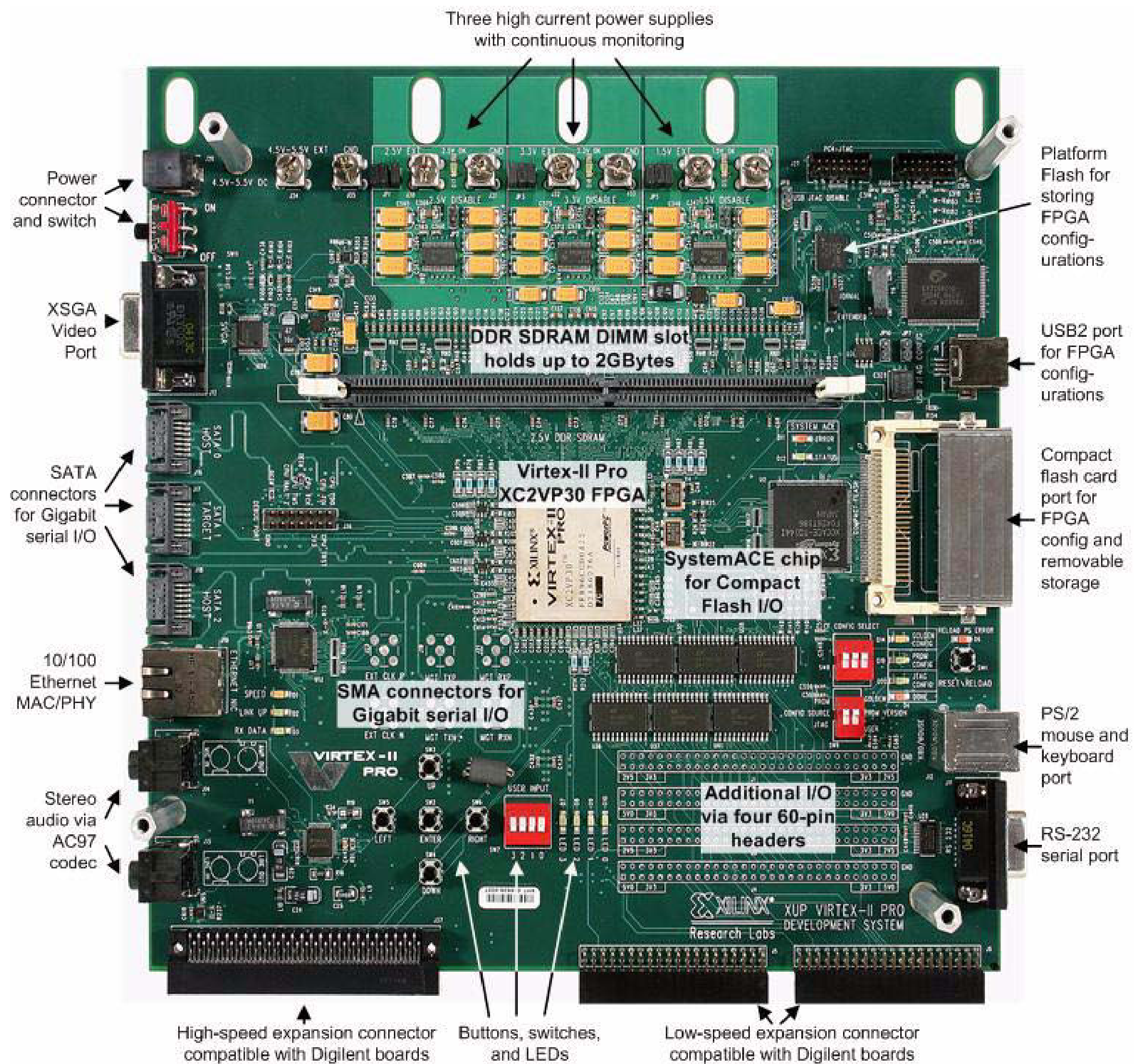


Figure E.6: XUP Virtex-II Pro Development System Board Photo [34]

E.3 Basic IP components

E.3.1 MicroBlaze Local Memory

Since the MicroBlaze soft core processor has a Harvard architecture it needs separated data and instruction memory. For the data and instruction memory a single block RAM (BRAM) [94] will be used since it is dual ported. Two physically separated and equally sized areas of the BRAM will be used for this purpose.

The dual ported block RAM is a special memory resource physically implemented inside the Xilinx FPGA (block RAM can be considered as a special kind of FPGA on chip memory, cp. Section 7.4). The block RAM is connected to the MicroBlaze processor via special point to point connection, the Local Memory Bus (LMB) [101, 102]. The LMB is a synchronous bus used primarily to access on-chip block RAM. It uses a minimum number of control signals and a simple protocol to ensure that local block RAM is accessed in a single clock cycle.

The size of the connected block RAM for local data and instruction is configurable from 2 KB up to 64 KB.

E.3.2 Interrupt Controller

Since the MicroBlaze has only one single interrupt port but there are possibly multiple interrupt sources among the peripherals and the user defined hardware blocks, it is essential to provide an interrupt controller. The used interrupt controller [107] is composed of a bus-centric wrapper containing the interrupt controller core and a bus interface. We make use of a simple, parametrized interrupt controller that, along with the appropriate bus interface, is attached to the OPB (On-chip Peripheral Bus).

Figure E.7 shows a block diagram of the interrupt controller which is organized into three functional units: interrupt detection and request generation, programmable registers and bus interface.

Interrupt detection can be configured for either level or edge detection for each interrupt input. If edge detection is chosen, synchronization registers are also included. Interrupt request generation is also configurable as either a pulse output for an edge sensitive request or as a level output that is cleared when the interrupt is acknowledged.

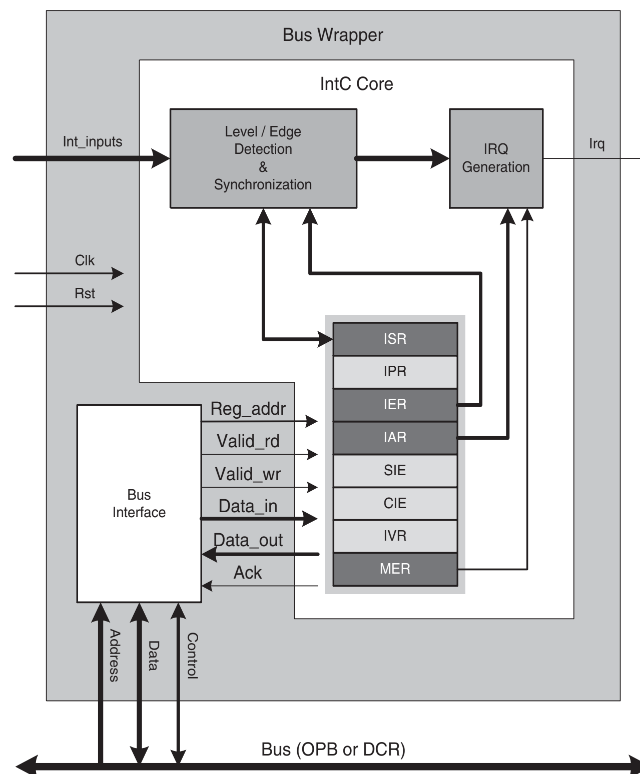


Figure E.7: Interrupt controller block diagram [107]

The interrupt controller contains the following programmer accessible registers [107]:

- Interrupt Status Register (ISR) is a read/write register that, when read, indicates which interrupt inputs are active (pre-enable bits). Writing to the ISR allows software to generate

interrupts until the Hardware Interrupt Enable (HIE) bit in the Master Enable Register (MER) has been enabled.

- Interrupt Pending Register (IPR) is a read only register that provides an indication of interrupts that are active and enabled (post enable bits). The IPR is an optional register and can be omitted to reduce FPGA resources required by an interrupt controller.
- Interrupt Enable Register (IER) is a read/write register whose contents are used to enable selected interrupts.
- Interrupt Acknowledge Register (IAR) is not an actual register. It is a write-only location used to clear interrupt requests.
- Set Interrupt Enables (SIE) is a write only location that provides the ability to set selected bits within the IER in one atomic operation, rather than requiring a read/modify/write sequence.
- Clear Interrupt Enables (CIE) is a write-only location that provides the ability to clear selected bits within the IER in a single atomic operation. Both SIE and CIE are optional and can be parametrized out of the design to reduce FPGA resource consumption by the interrupt controller.
- Interrupt Vector Register (IVR) is a read-only register that contains the ordinal value of the highest priority interrupt that is active and enabled. The IVR is optional and can be parametrized out of the design to reduce FPGA resources.
- Master Enable Register (MER) is a read/write, two-bit register used to enable or disable the IRQ output and to enable hardware interrupts (when hardware interrupts are enabled, software interrupts are disabled until the interrupt controller is reset).

The On-chip Peripheral Bus (OPB) interface provides a slave interface on the OPB for transferring data between the interrupt controller and the processor. The registers described above are memory mapped into the OPB address space and data transfers occur by using OPB byte enables.

The register addresses are fixed on four byte boundaries and the registers and the data transfers to and from them are always as wide as the data bus.

The number of interrupt inputs is configurable up to the width of the data bus, which is also set by a configuration parameter. The base address for the registers in the bus interface of the interrupt controller is set by a configuration parameter.

E.3.3 Timer

The Timer/Counter [109] is organized as two identical timer modules as shown in Figure E.8. Each timer module has an associated load register that is used to hold either the initial value for the counter for event generation, or a capture value, depending on the mode of the timer.

The generate value is used to generate a single interrupt at the expiration of an interval, or a continuous series of interrupts with a programmable interval. The capture value is the timer value that has been latched on detection of an external event. The clock rate of the timer modules is the clock of the OPB (no pre-scaling of the clock is performed). All of the Timer/Counter interrupts are OR'ed together to generate a single external interrupt signal. The interrupt service routine reads the control/status registers to determine the source of the interrupt.

E.3.4 Universal Asynchronous Receiver Transmitter (UART)

The UART [110] integrated in the `xilinx_microblaze` architecture building bock is usable for debugging purpose only. It is attached to the OPB as a slave and has several configurable features:

- OPB slave bus interface with byte-enable support
- One transmit and one receive channel (full duplex)

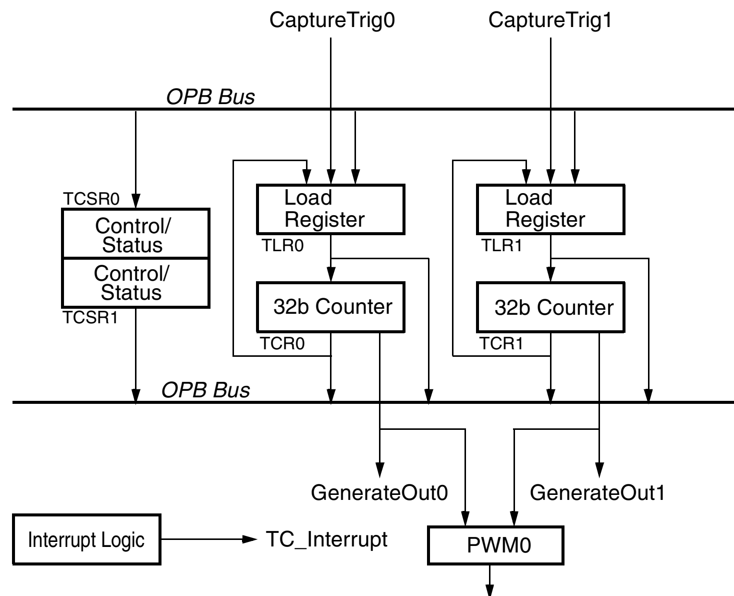


Figure E.8: Timer/Counter block diagram [109]

- 16-character transmit FIFO and 16-character receive FIFO
- Number of data bits in a character is configurable (5-8)
- Parity which can be turned on/off and can be configured as odd or even
- Configurable baud rate which is derived from the OPB clock

If interrupts are enabled, an interrupt is generated when one of the following conditions is true:

1. When there exists any valid character in the receive FIFO, the interrupt stays active until the receive FIFO is empty.

2. When the transmit FIFO goes from not empty to empty, such as when the last character in the transmit FIFO is transmitted, the interrupt is only active one clock cycle.

The "transmit buffer empty" interrupt is an edge interrupt and the "receive buffer empty" is a level interrupt.

E.3.5 Microprocessor Debug Module (MDM)

The Microprocessor Debug Module (MDM) [104] which can be shared between multiple Micor-Blaze processors enables JTAG-based debugging of one or more MicroBlaze processor cores. The most significant features of the MDM are:

- Support for JTAG-based software debug tools (e.g. the gdb)
- Support for debugging a configurable number (1-8) of MicroBlaze processors
- Support for synchronized control of multiple processors - stop and single step
- Support for a JTAG based UART with an OPB interface

E.3.6 On-Chip Peripheral Bus (OPB)

The IBM CoreConnect bus architecture can be partitioned into three subsystems: PLB (Processor Local Bus), OPB (On-Chip Peripheral Bus) and DCR (Device Control Register Bus). In the following we will present the supported OPB only. It is a general-purpose synchronous bus designed for connection of on-chip peripheral devices.

E.3.6.1 Features

The OPB includes the following features [113]:

- 32-bit or 64-bit data bus
- Up to 64-bit address
- Supports 8-bit, 16-bit, 32-bit, and 64-bit slaves
- Supports 32-bit and 64-bit masters
- Dynamic bus sizing with byte, half-word, full-word, and double-word transfers
- Optional Byte Enable support
- Distributed multiplexer bus instead of 3-state drivers
- Single cycle transfers between OPB master and OPB slaves (not including arbitration)
- Support for sequential address protocol
- 16-cycle bus time-out (provided by arbiter)
- Slave time-out suppress capability
- Support for multiple OPB bus masters
- Support for bus parking
- Support for bus locking
- Support for slave-requested retry
- Bus arbitration overlapped with last cycle of bus transfers

E.3.6.2 FPGA implementation supported features

The OPB is a full-featured bus architecture with many features that increase bus performance. Most of these features can be effectively used in an FPGA architecture. However, some features can result in the inefficient use of FPGA resources or can lower system clock rates. Consequently, Xilinx uses an efficient subset of the OPB for Xilinx developed OPB devices. Thus, the OPB provided by Xilinx is not fully OPB V2.1 compliant. A detailed specification of the OPB V2.1 can be found in [151].

Xilinx-developed OPB devices adhere to the following OPB usage rules [113]:

- The width of the OPB data buses and address buses is 32 bits. Note that some peripherals may parameterize these widths, but currently only 32-bit buses are supported. Peripherals that are smaller than 32-bits can be attached to the OPB with a corresponding restriction in addressing. For example, an 8-bit peripheral at base address A can be attached to byte lane 0, but can only be addressed at A, $A + 4$, $A + 8$, and so on.

- All OPB devices (masters and slaves) are byte-enable devices. These devices do not support the legacy data transfer signals and therefore do not support dynamic bus sizing. OPB masters do not mirror data to unused byte lanes.

- All OPB devices (masters and slaves) are required to output logic zero when they are inactive. This eliminates the need for the `Mn_DBusEn` and `Sln_DBusEn` signals external to the master or slave. The enable function is still implemented within the device.

- To obtain better timing in the FPGA implementation of the OPB, the `OPB_timeout` signal is registered. This means that all slaves must assert `Sl_xferAck` or `Sl_retry` on or before the rising edge of the 16th clock cycle after the assertion of `OPB_select`. If an OPB slave wishes to assert `Sl_toutSup`, `Sl_toutSup` must be asserted on or before the rising edge of the 15th clock after the assertion of `OPB_select`.

- The byte-enables and the least-significant address bits are driven by all masters and contain consistent information.

- All OPB slave devices that require a continuous address space (use of all byte lanes) will implement an attachment to the OPB bus that is as wide as the OPB data width, regardless of device width. This eliminates the need for left justification on the OPB bus and eliminates the need for masters to mirror write data.

- By convention, registers in all OPB slave devices are aligned to word boundaries (lowest two address bits are "00"), regardless of the size of the data in the register or the size of the peripheral.

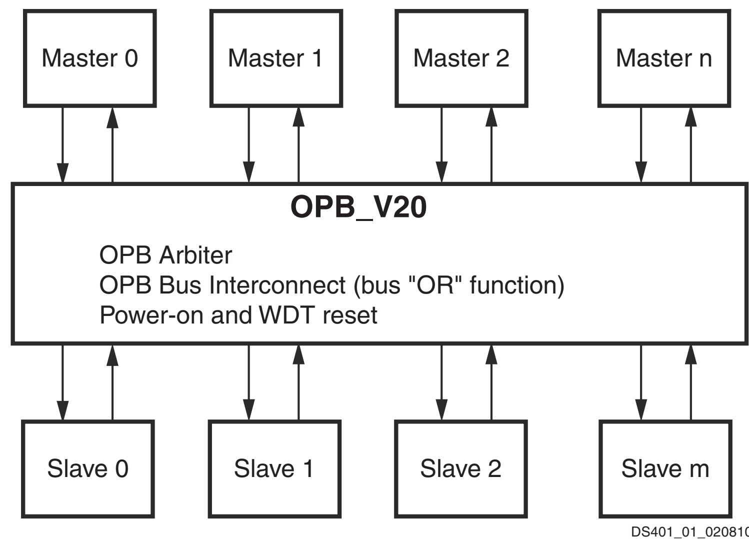


Figure E.9: Block diagram of an OPB with several attached master and slave components [105]

E.3.6.3 Connection

An OPB system is composed of masters, slaves, a bus interconnect, and an arbiter as shown in Figure E.9. The OPB_V20 [105] implements the bus interconnect and arbiter function in an OPB system. In Xilinx FPGAs, the OPB is implemented as a simple OR structure.

The OPB bus signals are created by logically OR'ing the signals that drive the bus. OPB devices that are not active during a transaction are required to drive zeros into the OR structure. This structure forms a distributed multiplexer and results in efficient bus implementations in FPGAs. Bus arbitration signals such as `M_request` and `OPB_MGrant` are directly connected between the OPB arbiter and each OPB master device.

The OPB_V20 supports up to 16 masters and an unlimited number of slaves (up to the hardware resources available). The port widths into the OPB_V20 are designed to increase in size as more masters or slaves are added. Xilinx recommends a maximum of 16 slaves on the OPB.

For example, the Master Data bus port has the width of the OPB data bus multiplied by the number of masters in the system. For a 32-bit OPB, Master 0 occupies `M_DBus(0:31)`, Master 1 occupies `M_DBus(32:63)`, and so on. Similarly for slaves, Slave 0 occupies `Sl_DBus(0:31)`, Slave 1 occupies `Sl_DBus(32:63)`, and so on. The 32-bit OPB data bus (`OPB_DBus`) is formed by OR'ing all the master and slave data buses together.

E.3.6.4 Arbitration

The OPB Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features [105]:

- Optional OPB slave interface (included in design via a design parameter)
- OPB Arbitration
  - arbitrates between 1-16 OPB Masters (the number of masters is parametrizeable)
  - arbitration priorities among masters programmable via register write
  - priority arbitration mode configurable via a design parameter
    - Fixed priority arbitration with processor access to read/write Priority Registers
    - Dynamic priority arbitration implementing a true least recent used (LRU) algorithm
- Two bus parking modes selectable via register write:
  - park on selected OPB master (specified in Control Register)
  - park on last OPB master which was granted OPB access

- Watchdog timer which asserts the OPB time-out signal if a slave response is not detected within 16 clock cycles.
- Registered or combinatorial Grant outputs configurable via a design parameter.

E.3.7 Intellectual Property Interface (IPIF)

The IPIF (Intellectual Property Interface) provides a standardized connection to the Xilinx Bus IP. The IPIF uses a back-end interface standard called the IPIC (IP Interconnect) which helps to connect the user logic to the IPIF services. The IPIF provides options which can be selected by the user, such as:
- Address decoding
- Interrupt management
- Software accessible registers
- IP reset via software-accessible registers
- Module identification register
- Read and write FIFOs between the user logic and the OPB
- Simple DMA capability for the read and transmit sides
- Scatter-Gather DMA (SG DMA) capability for the read and transmit sides

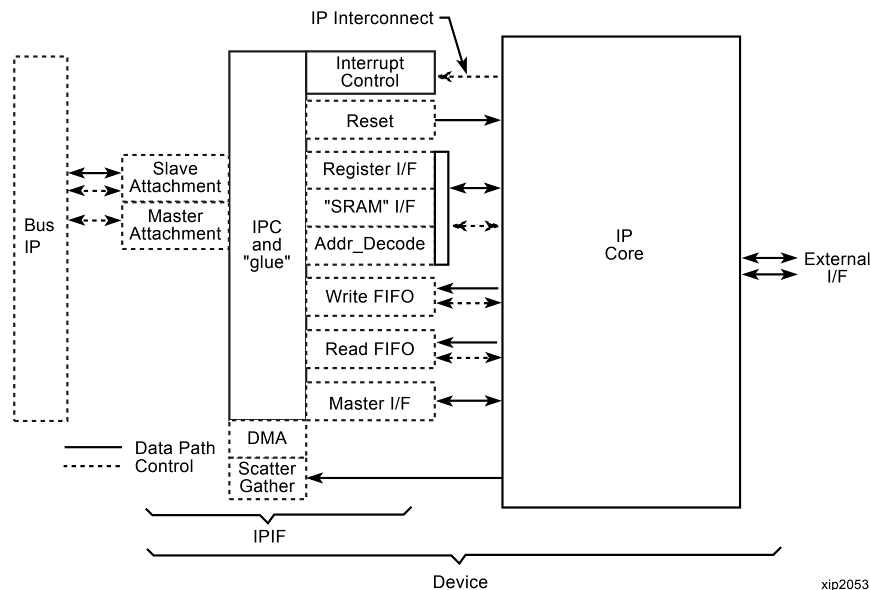


Figure E.10: A Bus (master/slave) IP using the full feature set of the IPIF [140]

Figure E.10 gives an overview of the IPIF features that can be used to connect a custom IP component with an available bus IP. The IPIF can be configured for master, slave and master/slave components.

For the connection of Shared Objects with bus IP, the minimal IPIF configuration, providing a slave bus interface and simple register access, is used (see Figure E.11).

Having a common interface, the IPIF reduces the development effort for custom bus cores. The IPIF is available for the Xilinx OPB [108] and PLB [112].

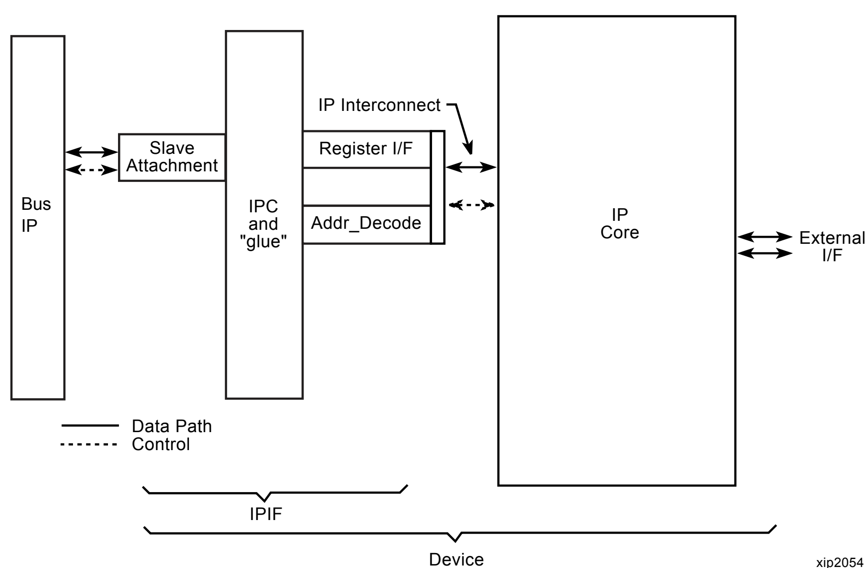


Figure E.11: A Bus (slave) IP using only the register feature of the IPIF [140]

APPENDIX F

Synthesis Subset[1]

In general an OSSS application layer design is synthesisable[2]. This section gives a short overview of the "really" synthesisable SystemC/OSSS language constructs accepted by the current implementation of the synthesis tool which will be referred to as *Fossy* or simply "the synthesiser".

F.1 Compatibility to the SystemC Synthesisable Subset

Fossy is quite compatible to the SystemC Synthesisable Subset. Table F.1 lists all important features of the synthesis subset. Unsupported features or features with restricted synthesis semantics are commented.

C++/SystemC™ feature			**Comment**
Translation units		√	Only one translation unit allowed.
Modules			
	Definition: `SC_MODULE`, `sc_module` inheritance	√	
	Members: signal, sub-module, ctors, `HAS_PROCESS`	√	
	Ports/Signals: `sc_signal`, `sc_in`, `sc_out`, `sc_inout`, `sc_in_clk`	√	
	Ports/Signals: `sc_out_clk`, `sc_inout_clk`	–	Not supported, but deprecated anyway.
	Ports/Signals: `*_resolved`, `*_rv`	–	Not supported.
	Ctor: w/ and w/o `SC_CTOR` macro	√	
	Deriving	√	
Datatypes			
	Integral types	√	
	Integral promotion, arith. conversion	√	
	Operators	√	
	Compounds: arrays, enums, class/struct/unions, functions	√	No pointers and no references supported.
	SysC: `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint`	√	

[1] This section is based on [44, Section 8.4].
[2] This should be no surprise since OSSS by definition is a synthesis subset.

	SysC: fixed-point types	–	Not supported.
	SysC: `sc_bv`	√	
	SysC: `sc_logic`	√	Converted to `sc_bv<1>` internally.
	SysC: `sc_lv`	–	Not supported.
	SysC: arithmetic operators	√	
	SysC: bitwise operators	√	
	SysC: relational oparators	√	
	SysC: shift operators	√	
	SysC: assignment operators	√	No chained assignments, LHS must be side effect-free.
	SysC: bit select operators	√	
	SysC: part select operators	√	Reverse ranges not supported.
	SysC: concatenation operators	√	Assignment to concats not supported.
	SysC: conversion to C integral (`to_int()`, `to_bool()` etc.)	√	
	SysC: additional methods: `iszero()`, `sign()`, `bit()`, `reverse()`, etc.	–	accepted, but not fully implemented
Declarations			
	typedef	√	
	enums	√	
	aggregates	–	Not supported. Reduced support for ROM initialization available.
	arrays	√	
	references	–	Copy-in copy-out support for functions/methods.
	pointers	–	Pointers allowed for sub-module instantiation.
Expressions		√	`new`/`delete`/pointers not supported, no chained assignments. `new` allowed for sub-module instantiation.
Functions		√	
Statements		√	
Processes		√	No `SC_THREAD` supported.
Sub-module instantiation		√	No support for positional port binding.
Namespaces		√	
Classes			
	Member functions	√	
	Member vars	√	
	Inheritance	√	
	Abstract classes	√	
	Constructors	√	No default arguments allowed.
Overloading		√	
Templates		√	
Pre-processing directives		√	

Table F.1: *Fossy* SystemC Synthesisable Subset support

F.2 Coding Guidelines

An OSSS design should be divided into several header (`.h`) and source files (`.cc`). Usually each header/source file pair should contain one module declaration/definition. An exception to this rule are templates. Templates should be completely described in the header file. A corresponding source file is not necessary (more precisely: not possible) in this case.

Limitation: Currently *Fossy* can only process a single translation unit, i.e. a single `.cc` file. As a workaround you can write a special main file, e.g. `fossy_main.cc` which includes all necessary `.cc` files. You should *not* `#include` any `.cc` files in header files.

An example of such a structure is shown in Listing F.1 and Listing F.2. The synthesis always needs a top-level **module** to start from, i.e. in order to do something useful with the synthesiser, the design must contain at least one module.

```
1  #include <systemc.h>
2  #include <osss.h>
3
4  SC_MODULE(SomeModule)
5  {
6    sc_in<bool>   somePort;
7
8    void someProcess();
9
10   SC_CTOR(SomeModule)
11   {
12     SC_METHOD(someProcess);
13     sensitive << somePort;
14   }
15 };
```

Listing F.1: General structure of the *Fossy* input, `SomeModule.h`

```
1  #include "SomeModule.h"
2
3  void
4  SomeModule::someProcess()
5  {
6    /* do something useful here */
7  }
```

Listing F.2: Source file `SomeModule.cc` corresponding to Listing F.1

F.2.1 Design Hierarchy

F.2.1.1 Modules

`SC_MODULE` is supported. An `SC_MODULE` can be defined via the macro `SC_MODULE` or by manually deriving from `sc_module`. If the latter form is used, the macro `SC_HAS_PROCESS(<ModuleName>)` has to inserted. An example is shown in Listing F.3.

```
1  #include <systemc.h>
2
3  SC_MODULE(myModule1)
4  {
5    // ...
6  };
7
8  struct myModule2 : public sc_module
9  {
10   // ...
11
12   // Required if this module contains processes
```

```
13   SC_HAS_PROCESS(myModule2);
14 };
```

Listing F.3: A synthesisable module definition

Limitation: Inheritance is not supported for user modules.

Note: Each module must have exactly one constructor.

F.2.1.2 Constructors

A module constructor can be defined via the macro `SC_CTOR` or it can be defined manually. An example is given in Listing F.4 and Listing F.5

```
1  // Begin of myModule1.h
2  #include <systemc.h>
3
4  SC_MODULE(myModule1)
5  {
6    sc_in<bool>   myPort;
7
8    // Alternative 1:
9    // Use SC_CTOR and place the body in the
10   // header file
11   SC_CTOR(myModule1)
12   : myPort("myPort")
13   , // ... more initialisers
14   {
15     /* constructor body */
16   }
17
18   // Alternative 2
19   // Manually specify the constructor and
20   // place the body in the header file
21   myModule(sc_module_name name)
22   : myPort("myPort")
23   , // ... more initialisers
24   {
25     /* constructor body */
26   }
27
28   // Alternative 3
29   // Manually specify the constructor and
30   // place the body in the source file
31   myModule(sc_module_name);
32   // the body is located in the .cc file
33
34   // Alternative 4:
35   // Use SC_CTOR and place the body in the
36   // source file
37   SC_CTOR(myModule1);
38   // the body is located in the .cc file
39   // (and looks exactly like the one for
40   // Alternative 3).
41
42 };
43 // End of myModule1.h
```

Listing F.4: Synthesisable module constructors, header file

```
1  // Begin of myModule1.cc:
2  #include "myModule1.h"
3
4    myModule1::myModule1(sc_module_name)
5    : myPort("myPort")
6    , // ... more initialisers
7    {
8      /* constructor body */
```

```
9   }
10 // End of myModule1.cc
```

Listing F.5: Synthesisable module constructors, source file

Limitation: Module constructors must have exactly one parameter: the module name, which must be of type `sc_module_name`.

Limitation: Module constructors must not contain complex control structures (if-then-else etc.). Simple (unrollable) for-loops are allowed.

Note: The name which is supplied as constructor argument to named objects like modules, ports, signals is ignored by *Fossy*. The resulting name will always be the attribute name.

F.2.1.3 Ports

The following ports are allowed:

- `sc_in<T>`
- `sc_out<T>`
- `sc_inout<T>`
- `osss_port_to_shared<IF>`

Ports may be used directly or by pointer. If the pointer variant is used, the port may be initialised in the intialiser list or in the constructor body as shown in Listing F.6.

```
1  #include <systemc.h>
2
3  SC_MODULE(Sub)
4  {
5    sc_in<bool>  port1,   // used directly
6                 *port2,   // by pointer
7                 *port3;   // by pointer
8
9    SC_CTOR(Sub)
10   : port1("port1")
11   , port2(new sc_in<bool>("port2"))   // initialised in the
12                                        //  initialiser   list
13   {
14     port3 = new sc_in<bool>("port3"); // assigned in the body
15   }
16 };
```

Listing F.6: Synthesisable ports

The type `T` may be any valid data type (see Section F.2.3).

The type `IF` may be any valid (user-defined) interface class.

Limitation: Arrays of ports are not supported. Structs containing ports are not supported.

F.2.1.4 Signals and Channels

The only synthesisable channel is `sc_signal<T>` where the type `T` may be any valid data type (see Section F.2.3).

The instantiation and naming requirements are exactly the same as in the case of ports (Section F.2.1.3).

Limitation: Structs containing signals are not supported.

F.2.1.5 Bindings

Every port of an instantiated module must be bound within the instantiating (parent) module.

Port bindings must be done via the `operator ()` as shown in the Listing F.7.

```
#include <systemc.h>

SC_MODULE(Bottom)
{
  sc_in<bool>         p1;
  sc_out<int>         p2;
  sc_in<sc_uint<8> >  p3;
};

SC_MODULE(Middle)
{
  sc_in<bool>         q1;
  Bottom b;

  sc_signal<int>  s;

  SC_CTOR(Middle)
  : b("b")
  {
    b.p1(q1);    // OK port-to-port binding
    b.p2(s);     // OK port-to-signal binding
                 // NOT OK: unbound b.p3
  }
};

SC_MODULE(Top)
{
  sc_in<sc_uint<8> >  r3;

  Middle m;

  SC_CTOR(Top)
  : m("m")
  {
    m.b.p3(r3); // NOT OK: bypasses the module hierarchy
  }
};
```

Listing F.7: Synthesisable bindings

Forbidden: Bindings must not bypass modules within the hierarchy.

Limitation: Positional binding is not supported.

F.2.2 Processes

The following process types are supported:

- `SC_METHOD`
- `SC_CTHREAD`

Constraints on `SC_CTHREAD`s:

- `SC_CTHREAD`s must not share member variables of a module. If two processes must exchange data either use a signal or a Shared Object. If a data member is exclusively used by a single process, better make it a local variable of the process body.

- `SC_CTHREAD`s must not terminate, i.e. an infinite loop is required in the process body. If you somehow need a terminating `SC_CTHREAD`, place a `while(true) wait();` at the end of the process body.

Constraints on `wait(...)` usage (in `SC_CTHREAD`s):

- Arbitrary `wait(n)` is allowed, i.e. `n` can be any (integer) expression. More specifically it is not required that `n` can be determined at compile-time. *WARNING:* If `n` cannot be determined at compile- time it is the user's responsibility to ensure that `n` never becomes zero (If this happens during normal SystemC simulation, the SystemC kernel will raise an error). *Fossy* won't (can't) check this and consequently won't raise an error. As a workaround, you could write `if (n) wait(n);`. In this case, however, you have to take special care for the following rules, because the `wait()` is conditional in this case. The constraints on conditional `wait()`s, however, *are* checked by *Fossy*.
- Loops must either
  1. always run into a `wait` in each iteration
  2. or have a fixed number of iterations which can be determined at compile time (This restriction is not due to OSSS, but due to the following synthesis tool).

Limitation: Fossy can only determine the number of iterations of `for`-loops.

Examples are shown in Listing F.8

Forbidden: `sensitive_pos` and `sensitive_neg` are not allowed. Use the corresponding `.pos()` and `.neg()` methods of the port.

Forbidden: The deprecated `watching(...)` is not allowed. Use `reset_signal_is(...)` instead.

```
1  #include <systemc.h>
2
3  SC_MODULE(Top)
4  {
5    sc_in<bool> clk,
6                reset;
7
8    sc_in<int>  px;
9
10   void myMethod();
11
12   // This CTHREAD is intended to show valid and invalid loop
13   // constructs (don't mind that the loops are actually unreachable).
14   void myCthread()
15   {
16     int x=4;
17     wait();
18
19     // Valid loop: always runs into a wait each iteration
20     while(true)
21     {
22       if (x--)
23         wait(1)
24       else
25         wait(2);
26     }
27
28     // Invalid loop: may do iterations without
29     // running into the wait;
30     while(true)
31     {
32       if (x--) wait();
33     }
34
35     // Valid loop: always runs into a wait each iteration
```

```
    // Note that the wait() be skipped entirely if the condition
    // is false before starting the loop.
    while(x--) wait();

    // Valid loop: always runs into a wait each iteration
    while(true)
    {
      if (x--) wait();

      "Some code";
      wait();
      "More code";
    }

    if(x==0) x=1;

    // OK: some x which is not zero
    wait(x);

    // Valid loop: always runs into a wait each iteration ...
    while(true)
    {
      if (x--) wait();

      // ... since this loop always causes a wait();
      for (int i=0;i<3;++i)
      {
        if (i==x-1) wait(); else wait();
      }
    }

    // Valid loop: always runs into a wait each iteration
    do
    {
      wait();
    } while (x--);

  }

  SC_CTOR(Top)
  {
    SC_METHOD(myMethod);
    sensitive << clk.pos()     // Note: .pos()/.neg() only works
              << reset;        //        for Boolean ports

    sensitive << px;           // OK: there may be multiple
                               //      sensitives

    SC_CTHREAD(myCthread(), clk.pos());
    reset_signal_is(reset, true);
  }
};
```

Listing F.8: Synthesisable processes

F.2.2.1 Effect of `wait()` usage on the number of states

In general the number of states of the resulting state machine equals the number of `wait()`s in the process body. A special case are loops: if there is a path through the loop body which does not contain any `wait()`s, *Fossy* tries to unroll that loop. Consequently the number of resulting states is the number of `wait()` statements[3] times the number of iterations. Please keep in mind that loops with many iterations which have to be unrolled due to non-optimal `wait()` usage, cause long synthesis runtimes, high memory consumption and finally a huge (but fast) circuit.

Another special case regarding the number of resulting states are `wait(n)`s. There are two different synthesis strategies. The first one is to simply unroll the `wait(n)` by `n` `wait(1)`s which

[3] In this case the `wait()`s will be conditional ones, because otherwise the loop would not have a path through its body without encountering a `wait()`.

results in `n` states. The second one is to replace the `wait(n)` by a loop with an unconditional `wait(1)` in the loop body. The resulting state machine of the second approach adds only one state to the state machine and one additional counter. The decision which of both approaches is chosen, depends on whether `n` is a compile-time known constant and also on its value.

F.2.2.2 Reset

The reset signal of an `SC_CTHREAD` is determined by the correpsonding `reset_signal_is(..)` statement in the constructor body. Please note that the reset of an `SC_CTHREAD` is always synchronous to the `SC_CTHREAD`'s clock. Consequently, *Fossy* always creates a state machine with a synchronous reset.

Note that the pre-reset behaviour before and after synthesis may differ. This is due to the fact the in the simulation an `SC_CTHREAD` always starts from the beginning of the method, whereas after synthesis (and in real hardware) the register which encodes the current state will start with a random value until it is reset. Starting with a random state in the context of an `SC_CTHREAD` would mean to start at some arbitrary `wait()`. In other words, the SystemC simulation with `SC_CTHREAD`s suggests that the state machine starts in its initial state even without reset, which is generally not the case in real hardware. However, after a reset the state machines before and after synthesis show exactly the same behaviour.

Since an `SC_CTHREAD` begins its execution from the beginning of the method body when the reset signal becomes activated, the reset state is determined by path(s) from the beginning of the method body to all potential first[4] `wait()`s. Note that it is not necessary to test the reset signal in the body of the process. A typical structure is shown in Listing F.9. Note that it is also valid to move the first `wait()` into the `while(true)` loop, because *Fossy* will detect that the loop will always be entered. Consequently it is possible to save one `wait()` and hence one state as shown in `myCthread2()`. The first variant, i.e. `myCthread()` will result in two equivalent states, both performing `x += px`.

```
1  #include <systemc.h>
2
3  SC_MODULE(Top)
4  {
5    sc_in<bool> clk,
6                reset;
7
8    sc_in<int>  px;
9
10   void myMethod();
11
12   void myCthread()
13   {             //
14     int x=4;    // Reset Part
15     wait();     //
16                    | State1
17     while(true)    |      +--+
18     {              |      |  |
19       x += px;     V      V  | State2
20       wait();                |
21                           |  |
22     }                     +--+
23   }
24
25   void myCthread2()
26   {             //
27     int x=4;    // Reset Part
28                 //
29     while(true) //  +--+
30     {           //  V  |
31       wait();   //     | State1
32       x += px;      |  |
33     }               +--+
34   }
35
```

[4]There may be more than one possible first `wait()` in the case of conditional `wait()`s in the reset part.

```
36   SC_CTOR(Top)
37   {
38     SC_CTHREAD(myCthread(), clk.pos());
39     reset_signal_is(reset , true);
40     SC_CTHREAD(myCthread2(), clk.pos());
41     reset_signal_is(reset , true);
42   }
43 };
```

Listing F.9: Reset part of an `SC_CTHREAD`

F.2.3 Datatypes

The following basic data types can be used for writing synthesisable models:

- `bool`
- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`,
- `int`, `unsigned int`,
- `long`, `unsigned long`,
- `long long`, `unsigned long long`,
- `sc_int<N>`, `sc_uint<N>`,
- `sc_bigint<N>`, `sc_biguint<N>`,
- `sc_bv<N>`,
- `sc_logic` (converted to sc_bv<1>),
- Enumeration types,

Currently not supported are:

- `sc_bit`
- `sc_lv<N>`,
- `sc_fixed<WL,IL, Q, O, n>`, `sc_ufixed<WL,IL, Q, O, n>`
- `sc_fixed_fast<WL,IL, Q, O, n>`, `sc_ufixed_fast<WL,IL, Q, O, n>`

Complex types like arrays, structures, classes and unions are synthesisable as long as they are constructed from synthesisable basic types. Classes, however, are regarded in greater detail in Section F.2.5.

All data types mentioned above can be used in the following places:

- Local variables in functions and processes
- Member variables
- Function parameters and member function parameters
- Signals (`sc_signal<T>`). Note: This requires `T` to define the `operator==(...)` and an `operator<<(...)` for stream insertion (and a `sc_trace(...)` function)
- Signal-ports (`sc_in<T>`, `sc_out<T>`, `sc_inout<T>`)
- `typedef`

Limitation: The resolved signal `sc_signal_rv<N>` and the corresponding ports `sc_in_rv<N>`, `sc_out_rv<N>` and `sc_inout_rv<N>` are currently not allowed for synthesisable models.

F.2.4 Statements and Expressions

The basic statements of C/C++ such as variable declaration and definition, assignments, control structures like `if () else`, `switch`, `for` and `while` loops are synthesisable. It is allowed to have `switch` statements with fall-through cases, i.e. cases without a `break` statement. Functions and function calls are synthesisable as long as they operate on synthesisable data types and consist of synthesisable constructs.

Parameters may be passed by value or by reference and may be `const` or non-`const`.

Forbidden: Functions and operators must not return references.

Limitation: Chaining multiple `operator =` in assignments (such as `a=b=c=d=0;` is not supported, yet.

Limitation: A `case`-part of a `switch`-statement which solely contains a `break;` is not supported. Workaround: insert an empty expression statement (`;`) right after the case label: `switch(i) { case 1: ; break; ... }`.

Limitation: A variable declaration in a `switch`-block before the first `case` label is not supported.

Limitation: Non-toplevel `case` labels are not supported.

Limitation: Variable declarations in `switch` clauses without a surrounding block are not supported.

Limitation: A non-void routine must not contain a `return` statement without an expression.

Limitation: Recursion is not supported.

Limitation: `sizeof(...)` is not supported.

Limitation: Only SystemC bitvector literals with prefix `"0b0"` are supported, e.g. `sc_int<3> x; x="0b0111"; x="0b0" "111";`

F.2.5 Classes and Inheritance

The following features of classes are allowed for synthesis:

- Non-`const` non-`static` data members
- `const` non-`static` data members
- `const static` data members
- Non-`static` member functions
- `static` member functions
- Virtual member functions (in conjunction with polymorphic objects)
- Pure virtual member functions (in conjunction with polymorphic objects)
- Base class(es) [Limitation: no non-virtual multiple inheritance from one base]
- Virtual base class(es) [Limitation: the virtual bases must not have any data members]
- Constructors [Limitation: copy constructor must have exactly one `const&` argument; default constructor must not have defaulted arguments]

- Member initialiser lists
- Overloaded operators
- User-defined implicit casts [Limitation: May collide with *Fossy*'s SystemC header files, especially the integer types]
- `explicit` constructors

Limitation: Copy assignment operators (`operator=`) must have the return type `void` and exactly one `const&` argument.

Limitation: If a user class contains an array member attribute, a copy constructor must be defined.

Limitation: Pointers to unused classes or template instances are not allowed. Note: This includes types like `sc_signal<>` which are actually templates.

Each data member must be of a synthesisable type (see Section F.2.3) and member functions must follow the same restrictions as functions do (see Section F.2.4).

Forbidden: Classes must not have an own thread of control, i.e. any `SC_METHOD`s, `SC_THREAD`s or `SC_CTHREAD`s. These are only allowed in modules.

F.2.6 Templates

Templates can be used to parameterise functions, classes/structs and member functions. Note that template classes may have template methods. Template specialisation and partial template specialisation are supported.

F.2.7 Namespaces

Namespaces are supported. The namespaces `osss`, `osss::synthesisable`, `sc_dt`, `sc_core` and `std` are reserved and must not be extended by the user.

F.2.8 Polymorphic Objects

Limitation: Polymorphic objects are not yet supported.

F.2.9 Shared Objects

Each guarded method must overwrite a virtual method from its interface class.

Parameters of guarded methods must be passed by value or const reference.

A guarded method which does not write any attribute must have a `wait()` in its body.

Limitation: The `schedule()` method of a scheduler must have a single `return` at the end of its body.

Limitation: Initialisation of schedulers must be performed in the constructor body, i.e. initialiser lists are currently ignored. This limitation also applies to all inherited constructors.

F.2.10 Non-Synthesisable

The following C++/SystemC/OSSS constructs are not synthesisable:

- OSSS architecture layer models
- Pointers (**Exception:** instantiation of modules, port and signals)
- Pointer arithmetics
- Member pointers
- Dynamic memory allocation with `new` and `delete`
- Placement-`new`
- Exceptions, throw-specifications
- Runtime `typeid`
- Reference types (**Exception:** function parameters)
- `dynamic_cast`
- Destructors (except for empty destructors)
- Static data members
- `mutable`, `volatile`
- `auto`, `register`
- `asm`
- `inline` (has no effect, does not harm)
- Standard C/C++ libraries with string handling, file I/O, ...
- Floating point arithmetic
- Bit fields (not needed – use SystemC data types instead)
- `friend` (partially implemented)
- `sc_time` (partially implemented)

APPENDIX G

Integrated Development Environment

G.1 Introduction

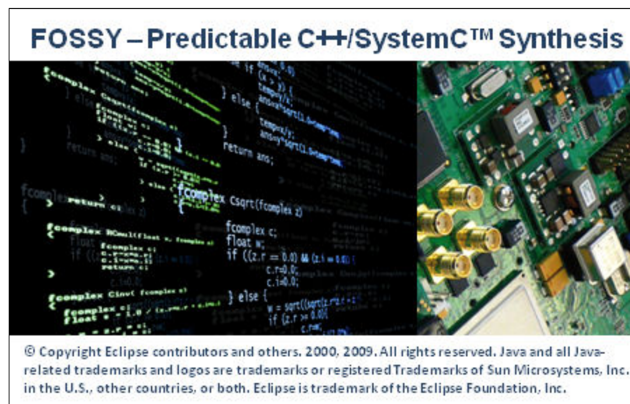


Figure G.1: *Fossy* IDE splash screen

The *Fossy* IDE (see Figure G.1) is based on the Eclipse CDT Environment and provides a tool suite for modeling, simulating, debugging and synthesis of OSSS and SystemC™ designs.

- Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and run-times for building, deploying and managing software across the life-cycle (see `http://www.eclipse.org/org/`)
- Eclipse CDT (Eclipse C/C++ Development Tooling) provides a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform. The features include: support for project creation and managed build for various tool chains, standard make build, source navigation, various source knowledge tools, such as type hierarchy, call graph, include browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation, source code refactoring and code generation, visual debugging tools, including memory, registers, and disassembly viewers (see `http://www.eclipse.org/cdt/`).

Our framework comes with a selection of different tools used for the development of software and custom hardware with OSSS and SystemC. Our framework consists of:

- *Fossy* (Functional Oldenburg System Synthesizer): A tool for converting OSSS and SystemC into synthesizable VHDL
- SystemC 2.2 simulation library from `http://www.systemc.org`
- Compiler Tool-chain & Build Environment
  - MinGW `http://www.mingw.org/`
  - MSYS `http://www.mingw.org/wiki/msys`
- GTKWave, a fully featured waveform viewer which reads LXT, LXT2, VZT, and GHW as well as standard Verilog VCD/EVCD files `http://gtkwave.sourceforge.net/`
- Eclipse IDE for C/C++ developers
  - Including VEditor plugin for Verilog and VHDL editing `http://veditor.wiki.sourceforge.net/`
  - with imported properties for SystemC, *Fossy* and GTKWave integration
  - with imported projects including example projects for *Fossy* OSSS and SystemC synthesis

G.2 Using the Eclipse CDT with FOSSY integration

G.2.1 Project Navigator

Figure G.2 shows the main window, with all available projects in the Navigator frame on the left side.

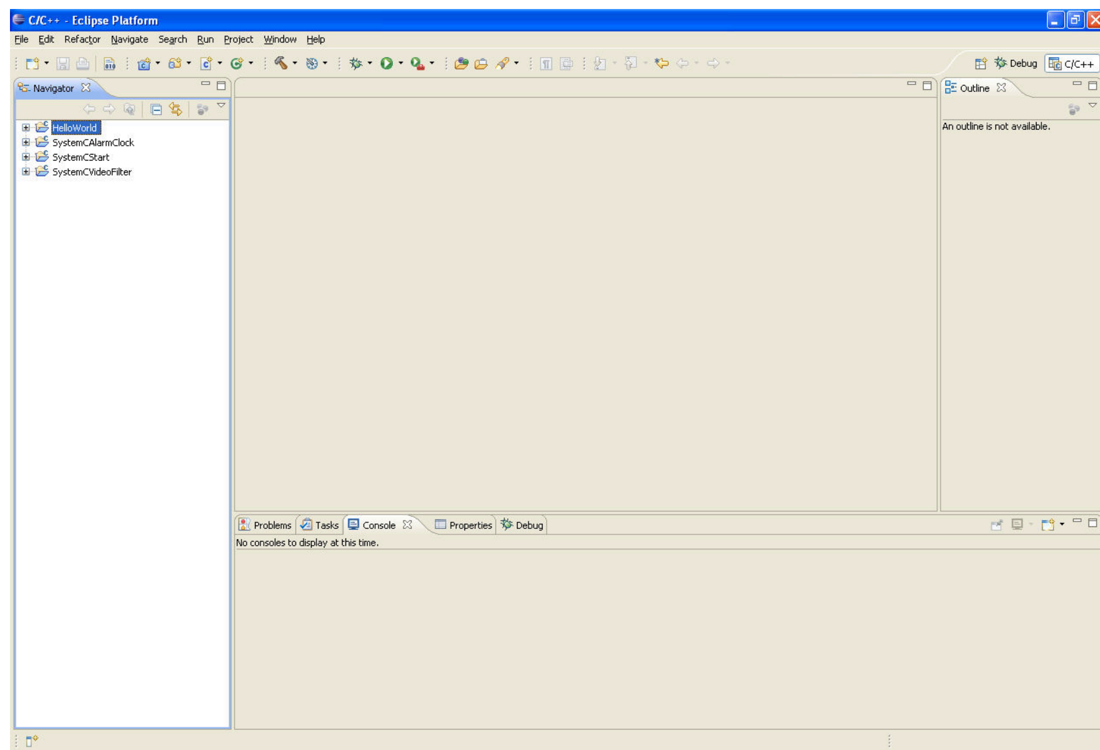


Figure G.2: *Fossy* IDE main window

G.2.2 The SystemC Alarm Clock Example

This example demonstrates the use of normal classes and objects in SystemC. A simple alarm clock is modeled with the aid of a Counter class which provide basic counter functionality. The alarm clock counts clock cycles and raises alarm for one clock cycle each time the specified period of time has passed. The simulation produces output on `stdout` as well as a waveform trace in VCD-format, showing clock, reset and alarm signals over time.

We demonstrate the usage of the *Fossy* IDE with the SystemC Alarm Clock project. To open the project just select it in the tree view in the project navigator window on the left side. Figure G.3 shows the `main.cc` of the SystemC Alarm Clock example in the editor window.

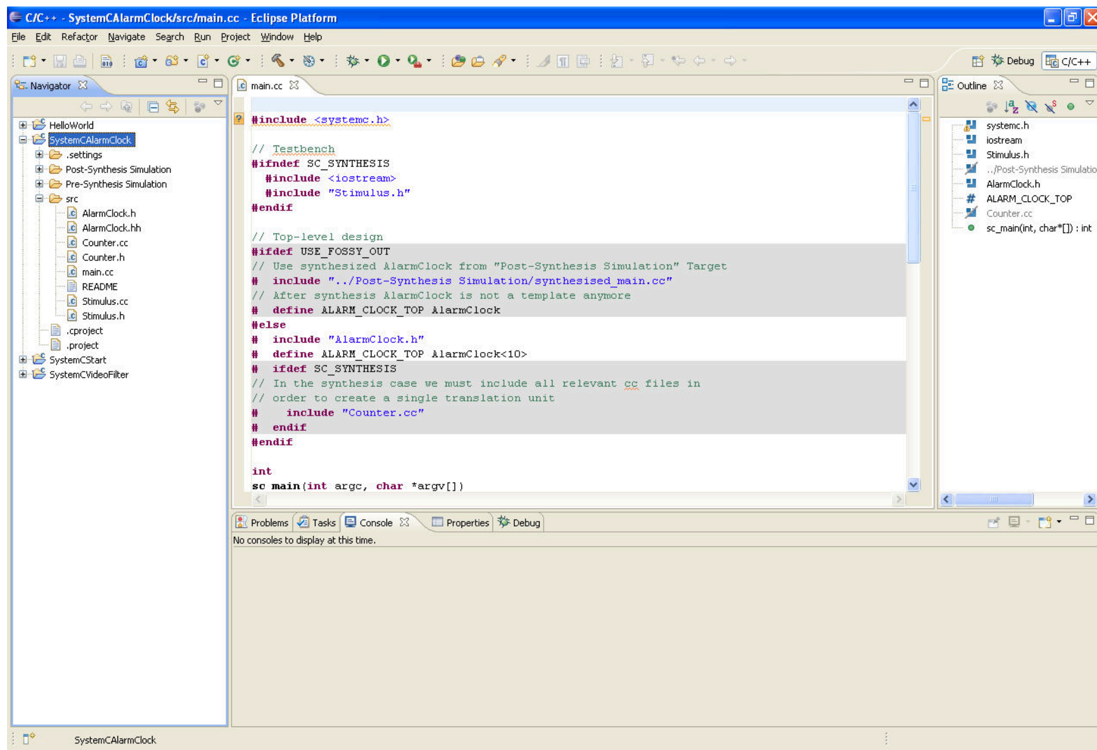


Figure G.3: `main.cc` of the SystemC Alarm Clock example in the editor window

Each project is structured by the following directories:

- **src:** Contains all relevant source files of the project. Open and edit a source file by double clicking in the project navigator tree view.

- **Pre-Synthesis Simulation:** This directory contains all files that are generated during the built of the Pre-Synthesis Simulation target. This target compiles the code and links is with the SystemC simulation kernel. The generated executable `TestSystemC.exe` performs the simulation of the user-specified alarm clock and generates the `trace.cvd` waveform.

- **Post-Synthesis Simulation:** This directory contains all files that are generated during the built of the Post-Synthesis Simulation target. This target performs the synthesis of the design by using *Fossy*. It takes the *Fossy* SystemC output and compiles and links is with the original testbench and the SystemC simulation kernel. The generated executable **TestSystemC.exe** performs the simulation of the user-specified alarm clock after *Fossy* synthesis and generates the `trace_fossy.cvd` waveform. *This simulation run show the design's behavior after* Fossy *synthesis. The behavior of the* Fossy *generated VHDL code is exactly the same.*

G.2.3 Building the Pre-Synthesis Model

For selecting the pre-synthesis simulation model make sure that the **SystemCAlarmClock** node in the project navigator tree is selected and click on "**Project** → **Build Configurations** → **Set Active** → **Pre-Synthesis Simulation**".

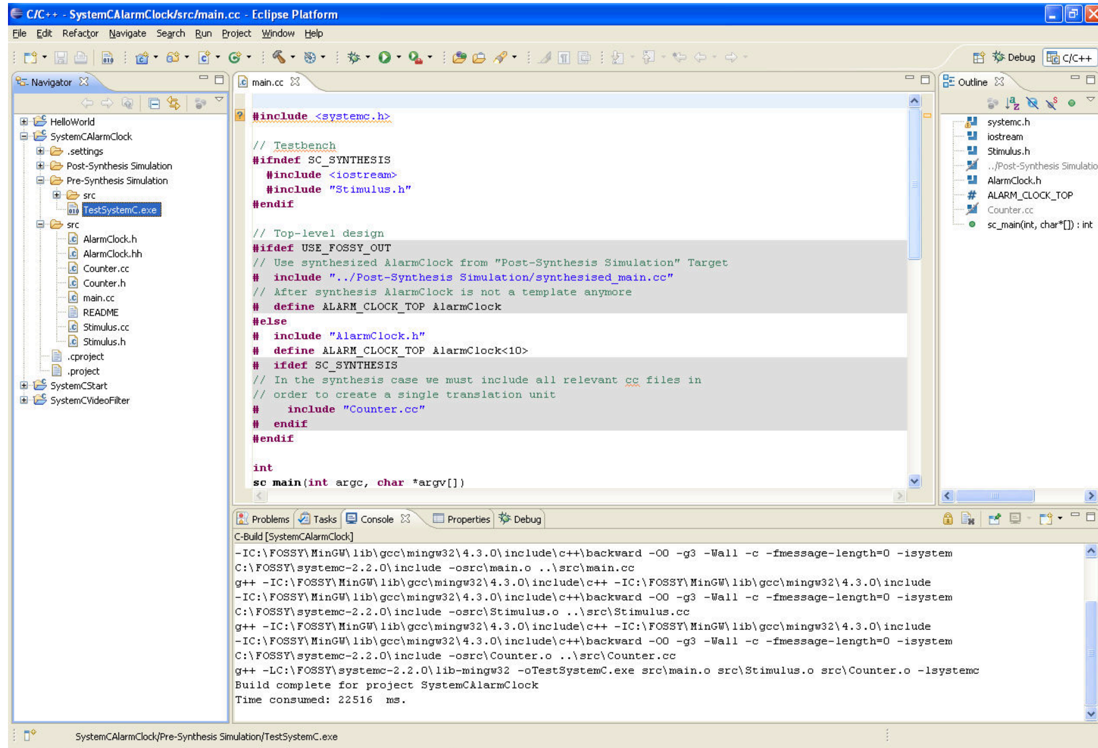


Figure G.4: Build process of the Pre-Synthesis Model

For building this project click on "**Project** → **Build Project**" Take a look at the console window at the bottom where the compiler and linker calls are shown (see Figure G.4).

For running the executable, open the `Pre-Synthesis Simulation` folder in the project navigator tree and select the file `TestSystemC.exe`. Then select "**Run → Run As Local C/C++ Application**".

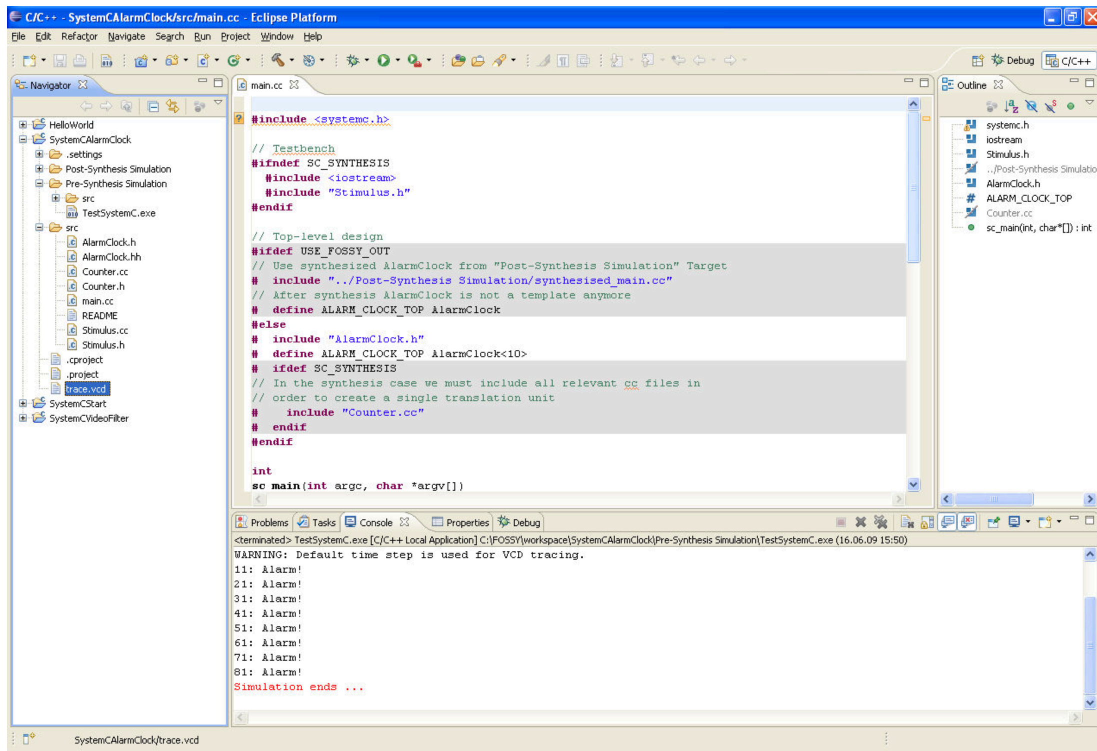


Figure G.5: Execution/Simulation of the Pre-Synthesis Model

Take a look at the console window at the bottom where the output of the SystemC simulation is shown. After simulation you can find the trace file `trace.vcd` at the bottom of the SystemC Alarm Clock project navigator tree (see Figure G.5).

We will now demonstrate how to watch this trace file using GTKWave.

For starting the waveform viewer on the trace file select "**Run → External Tools → GTKWave**". The window as shown in Figure G.6 appears.

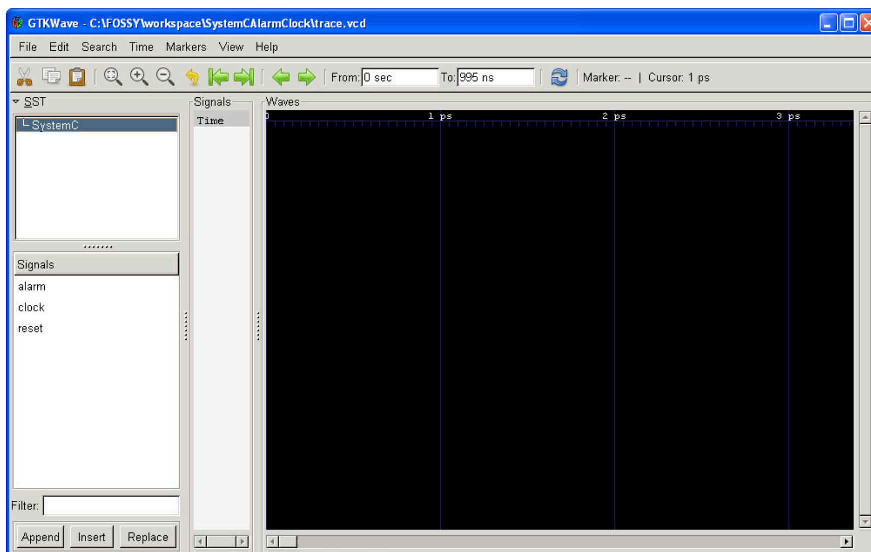


Figure G.6: Waveform viewer

Click on `SystemC` in the upper right window with the label SST (Signal Search Tree).

Afterwards the following signals should appear in the signals window below: `alarm`, `clock` and `reset`.

Select each signal and press the *Append* button to add them to the Waves viewer on the right side. After all signals have been added to the Waves viewer we need to zoom out by clicking on the magnifying glass item in the tool bar for several times.

As a result you should see the same window as in Figure G.7.

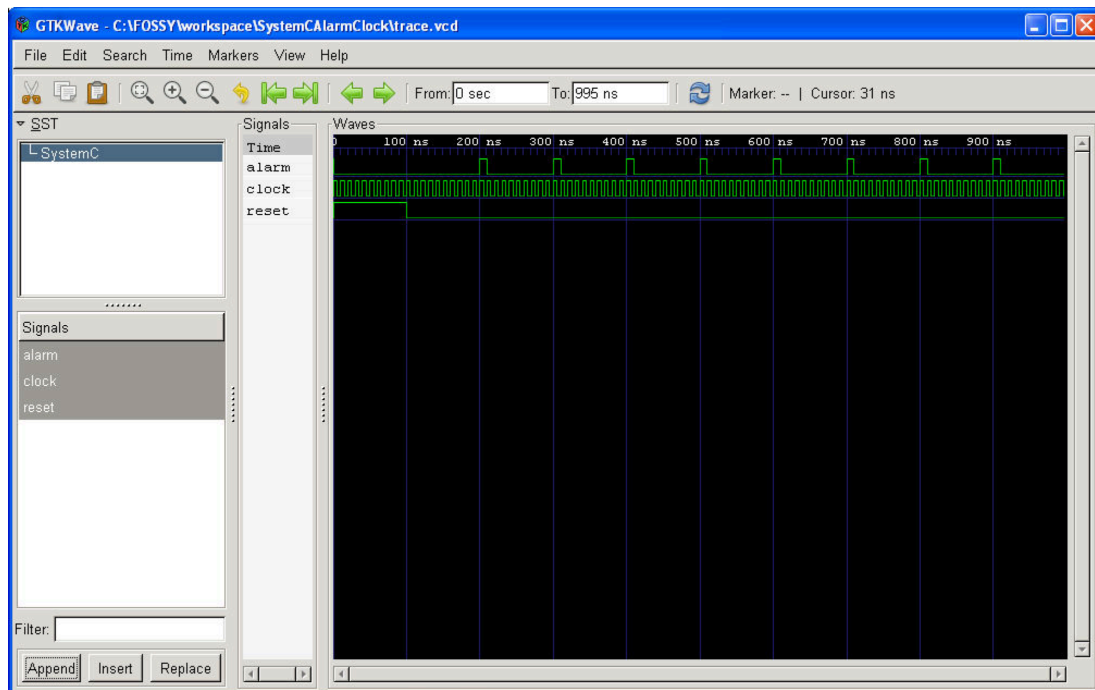


Figure G.7: Waveform viewer with simulation trace output

G.2.4 Building the Post-Synthesis Model

Next we want to perform *Fossy* synthesis of the Alarm Clock design.

We select the `SystemCAlarmClock` node in the project navigator tree and activate the post-synthesis simulation model by clicking on "**Project** → **Build Configurations** → **Set Active** → **Post-Synthesis Simulation**".

For building this project click on "**Project** → **Build Project**"

This should first start *Fossy* and then call the compiler tool chain including the SystemC output of *Fossy*. Take a look at the console window at the bottom to watch the *Fossy* synthesis and model compilation process.

The *Fossy* SystemC output can be examined by opening the Post-Synthesis Simulation folder in the project navigator tree and by double clicking on the file `synthesised_main.cc`.

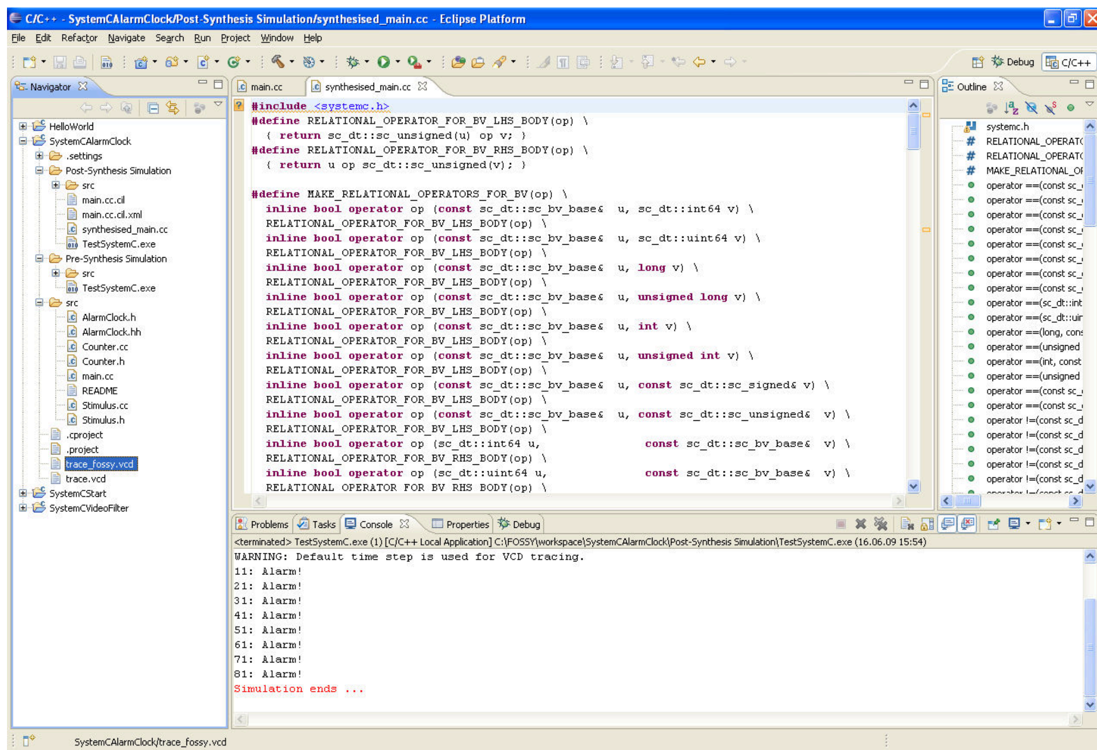


Figure G.8: Execution/Simulation of Post-Synthesis Model

For running the executable, open the `Post-Synthesis Simulation` folder in the project navigator tree and select the file `TestSystemC.exe`. Then select "**Run** → **Run As** → **Local C/C++ Application**". The console window at the bottom of Figure G.8 shows the output of the simulation. This is equal to the console output of the input model (see Figure G.4).

After simulation you can find the trace file `trace_fossy.vcd` at the bottom of the SystemC Alarm Clock project navigator tree. This file can be examined by GTKWave in the same way as shown above for the Pre-Synthesis target.

G.2.5 Generating VHDL code

This section demonstrates how to use the *Fossy* Framework to generate VHDL code for either simulation of Synthesis purpose with 3rd party tools like (Mentor Graphics ModelSim and Xilinx Synthesis Tool (XST))

For the SystemC Alarm Clock example navigate to the `src` directory in the project navigator tree and select the `main.cc` file.

For starting *Fossy* synthesis with VHDL output select "**Run** → **External Tools** → **FOSSY [VHDL]**"

This will start a *Fossy* run in the console window and generate a file named `synthesized_main.vhdl` in the `src` directory. You can view this file with a VHDL editor by double clicking on the file name.

The window shown in Figure G.9 shows the VHDL editor with its language specific Outline window on the right side. Since *Fossy* generated VHDL output is rather large the outline view helps navigating through the VHDL output.

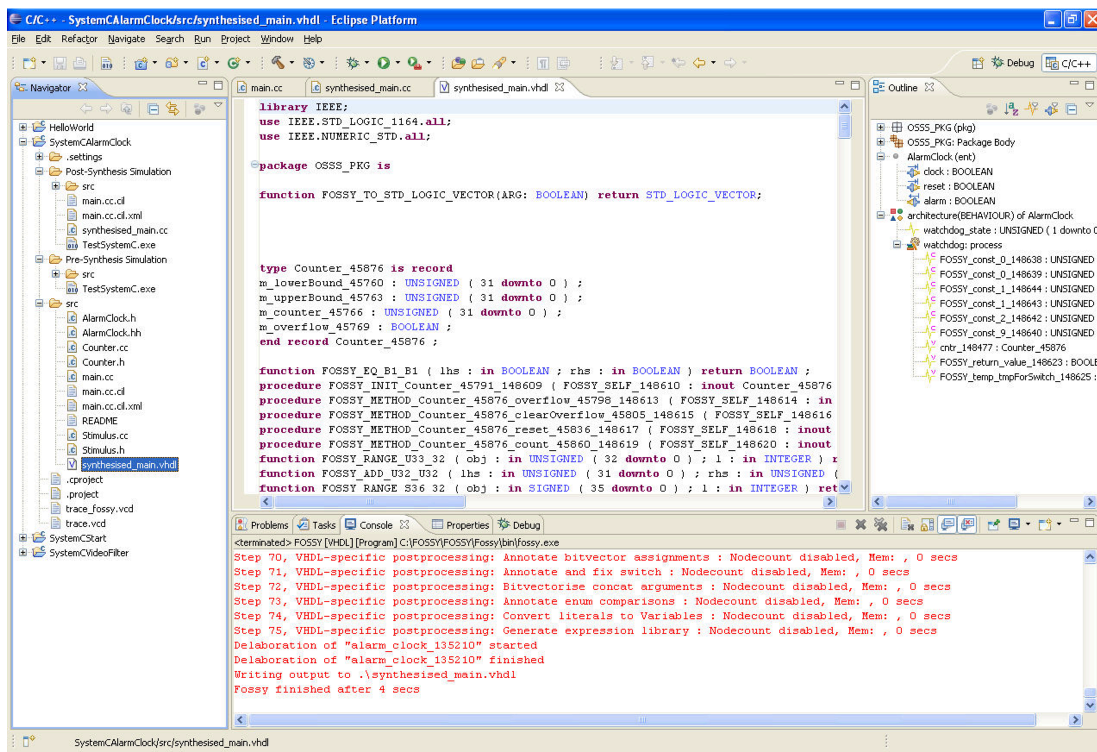


Figure G.9: VHDL synthesis output view

G.3 Creating a custom project

The recommended way of creating a custom projects is:

1. Copy the `SystemCStart` project by selecting it in the project navigator tree, pressing the right mouse button and select *Copy*
2. Then select the `SystemCStart` project and press the right mouse button again, but now select *Paste*
3. Enter the desired name of your new project in the pop-up window.
4. Now you are ready to start coding, simulation and synthesis.

APPENDIX H

OSSS Behavior Graphs

Figure H.1: Structure of the JPEG encoder OSSS Behavior Specification Model

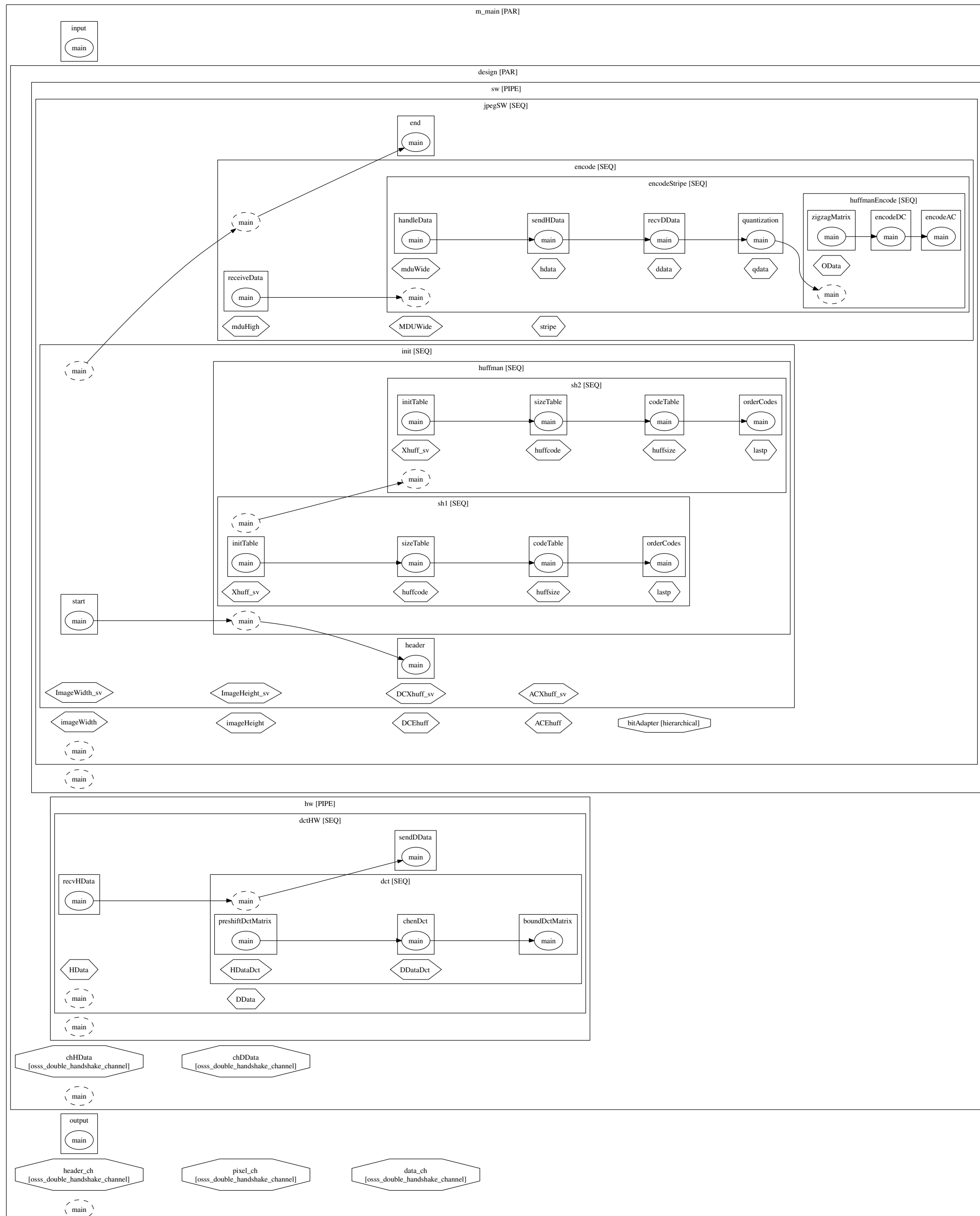


Figure H.2: Structure of the JPEG encoder OSSS Behavior Architecture Model

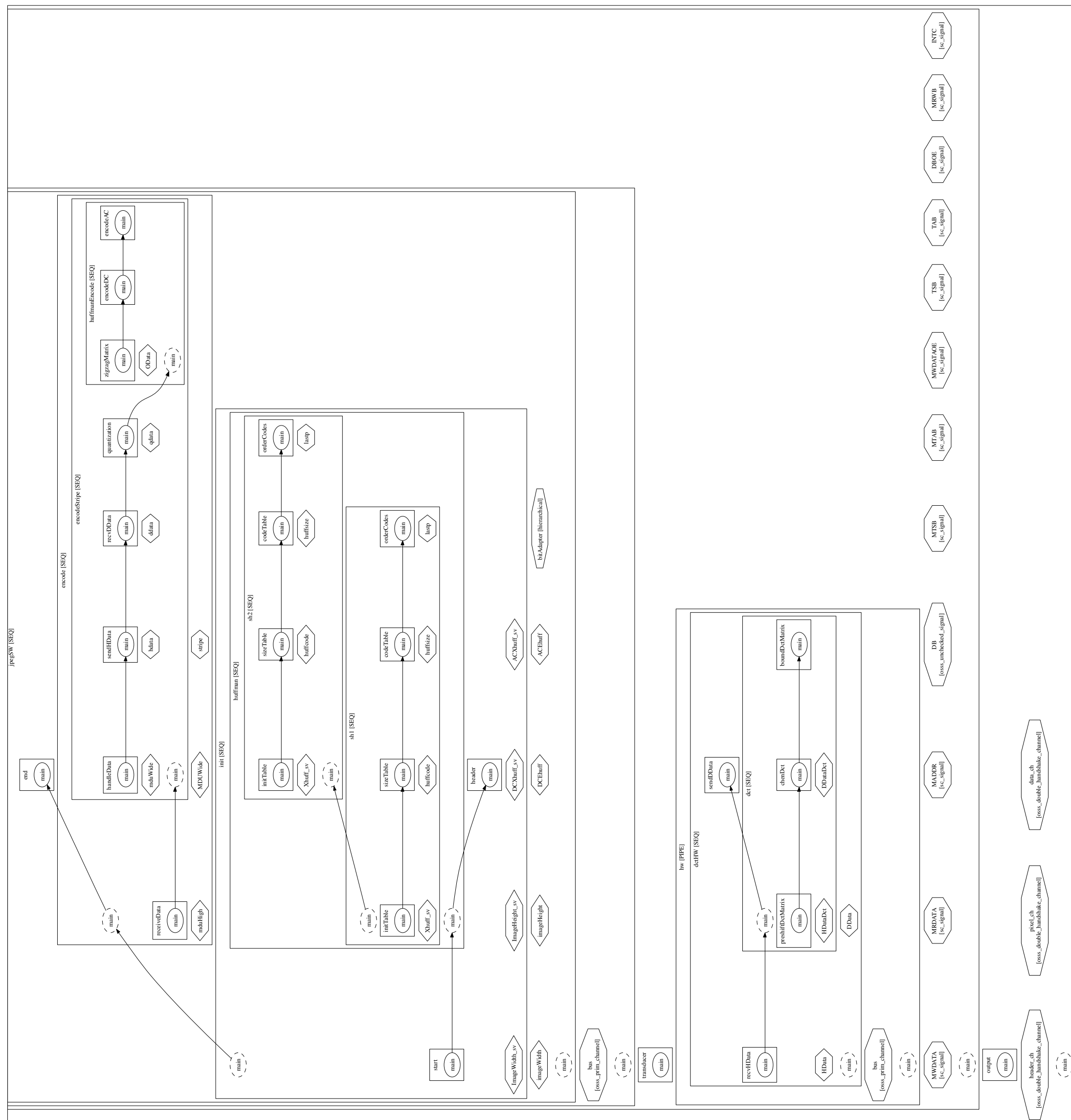


Figure H.3: Structure of the JPEG encoder OSSS Behavior communication model

Bibliography

Disclaimer: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Carl von Ossietzky Universität Oldenburg's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

[1] Tim Schmidt, Kim Grüttner, Rainer Dömer, and Achim Rettberg. "A Program State Machine Based Virtual Processing Model in SystemC". In: *The 4th Embedded Operating Systems Workshop (EWiLi'14)*. ACM SIGBED Review (ISSN: 1551–3688) Special Interest Group on Embedded Systems. Nov. 2014.

[2] *Vivado Design Suite – AXI Reference Guide, UG1037 (v1.0)*. Xilinx. 2014.

[3] *Vivado Design Suite – User Guide – High-Level Synthesis, UG902 (v2014.1)*. Xilinx. 2014.

[4] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. "Towards Performance Analysis of SDFGs Mapped to Shared-Bus Architectures Using Model-Checking". In: *DATE'13: Proceedings of the conference on Design, automation and test in Europe*. 2013.

[5] Kim Grüttner, Philipp A. Hartmann, Kai Hylla, Sven Rosinger, Wolfgang Nebel, Fernando Herrera, Eugenio Villar, Carlo Brandolese, William Fornaciari, Gianluca Palermo, Chantal Ykman-Couvreur, Davide Quaglia, Francisco Ferrero, and Raúl Valencia. "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration". In: *Microprocessors and Microsystems - Embedded Hardware Design* 37.8-C (2013), pp. 966–980.

[6] T. Mück and A Fröhlich. "Towards Unified Design of Hardware and Software Components Using C++". In: *Computers, IEEE Transactions on* PP.99 (2013), pp. 1–1. ISSN: 0018-9340. DOI: `10.1109/TC.2013.159`.

[7] Tim Schmidt. "A Program State Machine Based Virtual Processing Model in SystemC™". Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2013.

[8] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2013. ISBN: 0321563840.

[9] JesúS Barba, Fernando RincóN, Francisco Moya, Julio Daniel Dondo, and Juan Carlos LóPez. “A Comprehensive Integration Infrastructure for Embedded System Design”. In: *Microprocess. Microsyst.* 36.5 (July 2012), pp. 383–392. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2012.02.007. URL: http://dx.doi.org/10.1016/j.micpro.2012.02.007.

[10] Matthias Bücker, Kim Grüttner, Philipp A. Hartmann, and Ingo Stierand. “System Specification and Design Languages – Selected Contributions from FDL 2010”. In: Springer, Jan. 2012. Chap. Mapping of Concurrent Object-Oriented Models to Extended Real-Time Task Networks. ISBN: 978–1–4614–1426–1.

[11] Object Management Group. *CORBA Component Model 3.3 Specification*. Tech. rep. Version 3.3. Object Management Group, 2012. URL: http://www.omg.org/docs/formal/06-04-01.pdf.

[12] Kim Grüttner, Philipp A. Hartmann, Kai Hylla, Sven Rosinger, Wolfgang Nebel, Fernando Herrera, Eugenio Villar, Carlo Brandolese, William Fornaciari, Gianluca Palermo, Chantal Ykman-Couvreur, Davide Quaglia, Francisco Ferrero, and Raúl Valencia. “COMPLEX: COdesign and Power Management in PLatform-Based Design Space EXploration”. In: *DSD*. IEEE, 2012, pp. 349–358. ISBN: 978-1-4673-2498-4.

[13] Accellera Systems Initiative. “IEEE Standard for Standard SystemC Language Reference Manual”. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (Sept. 2012), pp. 1 –638. DOI: 10.1109/IEEESTD.2012.6134619.

[14] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[15] Rauf Salimi Khaligh and Martin Radetzki. “Semantics and efficient simulation of accuracy-adaptive TLMs”. English. In: *Design Automation for Embedded Systems* 16.3 (2012), pp. 1–29. ISSN: 0929-5585. DOI: 10.1007/s10617-012-9095-9. URL: http://dx.doi.org/10.1007/s10617-012-9095-9.

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 365–376. ISSN: 0163-5964. DOI: 10.1145/2024723.2000108. URL: http://doi.acm.org/10.1145/2024723.2000108.

[17] Philipp A. Hartmann, Kim Grüttner, Philipp Ittershagen, and Achim Rettberg. “A Framework for Generic HW/SW Communication using Remote Method Invocation”. In: *ESLsyn - The 2011 Electronic System Level Synthesis Conference*. ECSI. June 2011.

[18] *IP Processor Block RAM (BRAM) Block (v1.00a)*. Xilinx Inc. 2011.

[19] *LogiCORE IP XPS Multi-channel External Memory Controller (XPS MCH EMC) (3.01a)*. Xilinx Inc. 2011.

[20] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, DS083 (v5.0)*. Xilinx Inc. 2011.

[21] Matthias Büker, Kim Grüttner, Philipp A. Hartmann, and Ingo Stierand. “Mapping of Concurrent Object–Oriented Models to Extended Real–Time Task Networks”. In: *Forum on Specification & Design Languages (FDL)*. Sept. 2010.

[22] Kim Grüttner, Henning Kleen, Frank Oppenheimer, Achim Rettberg, and Wolfgang Nebel. “Towards a synthesis semantics for systemC channels”. In: *CODES+ISSS 2010*. Ed. by Tony Givargis and Adam Donlin. ACM, 2010, pp. 163–172. ISBN: 978-1-60558-905-3.

[23] Philipp A. Hartmann, Kim Grüttner, Achim Rettberg, and Ina Podolski. “Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC”. In: *7th IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*. Sept. 2010.

[24] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*. Xilinx Inc. 2010.

[25] *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)*. Xilinx Inc. 2010.

[26] *PowerPC Processor Reference Guide*. Xilinx Inc. 2010.

[27] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 1441905030, 9781441905031.

[28] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor P. Stefanov, Daniel D. Gajski, and Jürgen Teich. "Electronic system-level synthesis methodologies". In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 28.10 (Oct. 2009), pp. 1517–1530. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2026356. URL: http://dx.doi.org/10.1109/TCAD.2009.2026356.

[29] Kim Grüttner, Frank Oppenheimer, Wolfgang Nebel, Jan Freuer, and Joachim Gerlach. "Rapid Prototyping und Synthese eines videobasierten Fahrerassistenzsystems mit C++ und SystemC(TM)". In: *10. Braunschweiger Symposium AAET 2009 – Automatisierungs–, Assistenzsysteme und eingebettete Systeme für Transportmittel*. Feb. 2009.

[30] Kecheng Hao and Fei Xie. "Componentizing hardware/software interface design". In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09*. 2009, pp. 232–237. DOI: 10.1109/DATE.2009.5090663.

[31] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. "SystemCoDesigner: an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications". In: *ACM Trans. Des. Autom. Electron. Syst.* 14.1 (Jan. 2009), 1:1–1:23. ISSN: 1084-4309. DOI: 10.1145/1455229.1455230. URL: http://doi.acm.org/10.1145/1455229.1455230.

[32] Rauf Salimi Khaligh and Martin Radetzki. "Adaptive Interconnect Models for Transaction-Level Simulation". English. In: *Languages for Embedded Systems and their Applications*. Ed. by Martin Radetzki. Vol. 36. Lecture Notes in Electrical Engineering. Springer Netherlands, 2009, pp. 149–165. ISBN: 978-1-4020-9713-3. DOI: 10.1007/978-1-4020-9714-0_10. URL: http://dx.doi.org/10.1007/978-1-4020-9714-0_10.

[33] Synthesis Working Group Members of the Open SystemC Initiative. *SystemC Synthesizable Subset, Draft 1.3*. Whitepaper. Open SystemC Initiative (OSCI), 2009.

[34] *Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual, UG069 (v1.2)*. Xilinx Inc. 2009.

[35] J. Cong. "A new generation of C-base synthesis tool and domain-specific computing". In: *SOC Conference, 2008 IEEE International*. 2008, pp. 386–386. DOI: 10.1109/SOCC.2008.4641556.

[36] *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*. IEEE, 2008. ISBN: 978-3-9810801-3-1.

[37] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. "System-on-chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design". In: *EURASIP J. Embedded Syst.* 2008 (Jan. 2008), 5:1–5:13. ISSN: 1687-3955. DOI: 10.1155/2008/647953. URL: http://dx.doi.org/10.1155/2008/647953.

[38] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and Shuvra S. Bhattacharyya. "A Generalized Static Data Flow Clustering Algorithm for Mpsoc Scheduling of Multimedia Applications". In: *Proceedings of the 8th ACM International Conference on Embedded Software*. EMSOFT '08. Atlanta, GA, USA: ACM, 2008, pp. 189–198. ISBN: 978-1-60558-468-3. DOI: 10.1145/1450058.1450084. URL: http://doi.acm.org/10.1145/1450058.1450084.

[39] G. Gailliard, H. Balp, C. Jouvray, and F. Verdier. "Towards a common HW/SW interface-centric and component-oriented specification and design methodology". In: *Forum on Specification, Verification and Design Languages, 2008. FDL 2008*. 2008, pp. 31–36. DOI: 10.1109/FDL.2008.4641417.

[40] Grégory Gailliard, Hugues Balp, Michel Sarlotte, and François Verdier. “Mapping Semantics of CORBA IDL and GIOP to Open Core Protocol for Portability and Interoperability of SDR Waveform Components”. In: *DATE*. IEEE, 2008, pp. 330–335. ISBN: 978-3-9810801-3-1.

[41] K. Gruttner, F. Oppenheimer, and W. Nebel. “OSSS methodology - system-level design and synthesis of embedded HW/SW systems in C++”. In: *Applied Sciences on Biomedical and Communication Technologies, 2008. ISABEL '08. First International Symposium on*. 2008, pp. 1–5. DOI: 10.1109/ISABEL.2008.4712587.

[42] Kim Grüttner. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – The OSSS 2.2.0 Tutorial*. OFFIS - Institute for Information Technology. Sept. 2008.

[43] Kim Grüttner and Wolfgang Nebel. “Modelling Program-State Machines in SystemC”. In: *FDL*. IEEE, 2008, pp. 7–12. ISBN: 978-1-4244-2265-4.

[44] Kim Grüttner, Andreas Herrholz, Philipp A. Hartmann, Andreas Schallenberg, and Claus Brunzema. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – The OSSS 2.2.0 Manual*. OFFIS - Institute for Information Technology. Sept. 2008.

[45] Kim Grüttner, Frank Oppenheimer, Wolfgang Nebel, Fabien Colas-Bigey, and Anne-Marie Fouilliart. “SystemC-based Modelling, Seamless Refinement, and Synthesis of a JPEG 2000 Decoder”. In: *DATE*. IEEE, 2008, pp. 128–133. ISBN: 978-3-9810801-3-1.

[46] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. “PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems”. In: *ACM Trans. Des. Autom. Electron. Syst.* 12.3 (May 2008), 24:1–24:25. ISSN: 1084-4309. DOI: 10.1145/1255456.1255461. URL: http://doi.acm.org/10.1145/1255456.1255461.

[47] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. “Behavioral partitioning with exploiting function-level parallelism”. In: *SoC Design Conference, 2008. ISOCC '08. International*. Vol. 01. 2008, pp. I–121–I–124. DOI: 10.1109/SOCDC.2008.4815588.

[48] P.A. Hartmann, H. Kleen, P. Reinkemeier, and W. Nebel. “Efficient modelling and simulation of embedded software multi-tasking using SystemC and OSSS”. In: *Forum on Specification, Verification and Design Languages, 2008 (FDL 2008)*. 2008, pp. 19–24. DOI: 10.1109/FDL.2008.4641415.

[49] Wolfgang Klingauf. “Systematic transaction level communication modeling with systemC”. PhD thesis. 2008, pp. 1–179.

[50] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. “A Retargetable Parallel-programming Framework for MPSoC”. In: *ACM Trans. Des. Autom. Electron. Syst.* 13.3 (July 2008), 39:1–39:18. ISSN: 1084-4309. DOI: 10.1145/1367045.1367048. URL: http://doi.acm.org/10.1145/1367045.1367048.

[51] *MISRA-C: 2008 - Guidelines for the use of the C++ language in critical systems*. MIRA Limited. 2008.

[52] H. Nikolov, T. Stefanov, and E. Deprettere. “Systematic and Automated Multiprocessor System Design, Programming, and Implementation”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.3 (2008), pp. 542–555. ISSN: 0278-0070. DOI: 10.1109/TCAD.2007.911337.

[53] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. “Daedalus: Toward Composable Multimedia MP-SoC Design”. In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 574–579. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391615. URL: http://doi.acm.org/10.1145/1391469.1391615.

[54] M. Radetzki and R.S. Khaligh. “Accuracy-Adaptive Simulation of Transaction Level Models”. In: *Design, Automation and Test in Europe, 2008. DATE '08*. 2008, pp. 788–791. DOI: 10.1109/DATE.2008.4484912.

[55] W. Wolf, A.A. Jerraya, and G. Martin. "Multiprocessor System-on-Chip (MPSoC) Technology". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.10 (2008), pp. 1701 –1713. ISSN: 0278-0070. DOI: `10.1109/TCAD.2008.923415`.

[56] C. Zebelein, J. Falk, C. Haubelt, and J. Teich. "Classification of General Data Flow Actors into Known Models of Computation". In: *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on.* 2008, pp. 119–128. DOI: `10.1109/MEMCOD.2008.4547699`.

[57] *128-Bit Processor Local Bus Architecture Specifications.* IBM Inc. 2007.

[58] Jesus Barba, Fernando Rincon, Francisco Moya, Felix J. Villanueva, David Villa, Julio Dondo, and Juan C. Lopez. "OOCE: Object-Oriented Communication Engine for SoC Design". In: *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 296–302. ISBN: 0-7695-2978-X. DOI: `http://dx.doi.org/10.1109/DSD.2007.86`.

[59] Bishnupriya Bhattacharya, John Rose, and Stuart Swan. "Language Extensions to SystemC: Process Control Constructs". In: *DAC.* 2007, pp. 35–38.

[60] Alan Burns and Andy J. Wellings. *Concurrent and real-time programming in Ada.* Cambridge University Press, 2007, pp. I–XIV, 1–461.

[61] Mark Burton, James Aldis, Robert Günzel, and Wolfgang Klingauf. "Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified". In: *FDL.* ECSI, 2007, pp. 92–97.

[62] Fabien Colas-Bigey, Giovanna Ferrara, Jan Freuer, and Joachim Gerlach. *Report on the Evaluation of the Language, Tools and Prototypes Developed in ICODES.* ICODES Deliverable D39 (public). OFFIS Institute for Information Technology, 2007.

[63] A. Gerstlauer, Dongwan Shin, Junyu Peng, R. Domer, and D.D. Gajski. "Automatic Layer-Based Generation of System-On-Chip Bus Communication Models". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 26.9 (2007), pp. 1676–1687. ISSN: 0278-0070. DOI: `10.1109/TCAD.2007.895794`.

[64] Jens Gladigau, Christian Haubelt, Bernhard Niemann, and Jürgen Teich. "Mapping Actor-Oriented Models to TLM Architectures". In: *Forum on specification and Design Languages (FDL'07).* 2007, pp. 128–133.

[65] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer, and Wolfgang Nebel. "Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS". In: *Proceedings of the SASIMI 2007.* 2007.

[66] Henning Kleen. "Effizienzanalyse methodenbasierter Hardware/Software Kommunikation aus Synthesesicht". MA thesis. Carl von Ossietzky Universität Oldenburg, Fakultät II - Department für Informatik, Abteilung Eingebettete Hardware-/Software-Systeme, 2007.

[67] W. Klingauf, R. Günzel, and C. Schröer. "Embedded software development on top of transaction-level models". In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on.* 2007, pp. 27–32.

[68] M. Montoreano. *Transaction Level Modeling using OSCI TLM 2.0.* Tech. rep. 2007.

[69] Bernhard Niemann and Christian Haubelt. "Towards a Unified Execution Model for Transactions in TLM". In: *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign.* MEMOCODE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 103–112. ISBN: 1-4244-1050-9. DOI: `10.1109/MEMCOD.2007.371237`. URL: `http://dx.doi.org/10.1109/MEMCOD.2007.371237`.

[70] Open Core Protocol International Partnership (OCP-IP). *A SystemC OCP Transaction Level Communication Channel.* Tech. rep. Beaverton, USA, 2007.

[71] Alberto Sangiovanni-Vincentelli. “Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design”. In: *Proceedings of the IEEE* 95.3 (2007), pp. 467–506. URL: http://chess.eecs.berkeley.edu/pubs/263.html.

[72] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. “A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs”. In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’07. Salzburg, Austria: ACM, 2007, pp. 9–14. ISBN: 978-1-59593-824-4. DOI: 10.1145/1289816.1289823. URL: http://doi.acm.org/10.1145/1289816.1289823.

[73] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. “Pn: A Tool for Improved Derivation of Process Networks”. In: *EURASIP J. Embedded Syst.* 2007.1 (Jan. 2007), pp. 19–19. ISSN: 1687-3955. DOI: 10.1155/2007/75947. URL: http://dx.doi.org/10.1155/2007/75947.

[74] *VPEP - Video Processing Evaluation Platform (Designers Guide)*. AE/EIP5, Robert Bosch GmbH. 2007.

[75] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2nd ed. Prentice Hall, Sept. 2006. ISBN: 0321486811. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0321486811.

[76] *Device Control Register Bus 3.5 Architecture Specifications, SA14-2706-03*. IBM Inc. 2006.

[77] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer, and Wolfgang Nebel. “Modelling and Synthesis of Communication Using OSSS-Channels”. In: *MBMV*. Ed. by Bernd Straube and Martin Freibothe. Fraunhofer Institut für Integrierte Schaltungen, 2006, pp. 38–47.

[78] Kim Grüttner, Cornelia Grabbe, Thorsten Schubert, Claus Brunzema, and Frank Oppenheimer. “OSSS-Channels: Modelling and Synthesis of Communication”. In: *Forum on specification and Design Languages, FDL 2006, September 19-22, 2006, Darmstadt, Germany, Proceedings*. ECSI, 2006, pp. 327–335. URL: http://www.ecsi-association.org/ecsi/main.asp?l1=library&fn=def&id=393.

[79] Stefan Ihmor. “Modeling and automated synthesis of reconfigurable interfaces”. PhD thesis. Fakultät für Elektrotechnik, Informatik und Mathematik, Institut für Informatik, Universität Paderborn, 2006.

[80] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. “UML-based Multiprocessor SoC Design Framework”. In: *ACM Trans. Embed. Comput. Syst.* 5.2 (May 2006), pp. 281–320. ISSN: 1539-9087. DOI: 10.1145/1151074.1151077. URL: http://doi.acm.org/10.1145/1151074.1151077.

[81] Wolfgang Klingauf, Hagen Gädke, and Robert Günzel. “TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC”. In: *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*. DATE ’06. Munich, Germany: European Design and Automation Association, 2006, pp. 1318–1323. ISBN: 3-9810801-0-6. URL: http://dl.acm.org/citation.cfm?id=1131481.1131843.

[82] Wolfgang Klingauf, Robert Günzel, Oliver Bringmann, Pavel Parfuntseu, and Mark Burton. “GreenBus: A Generic Interconnect Fabric for Transaction Level Modelling”. In: *Proceedings of the 43rd Annual Design Automation Conference*. DAC ’06. San Francisco, CA, USA: ACM, 2006, pp. 905–910. ISBN: 1-59593-381-6. DOI: 10.1145/1146909.1147139. URL: http://doi.acm.org/10.1145/1146909.1147139.

[83] ARM Ltd. *AMBA 3 AHB-Lite Protocol v1.0 Specification*. Datasheet. 2006.

[84] M. Mitic and M. Stojcev. *An overview of on-chip buses*. Tech. rep. Facta Universitatis, 2006.

[85] *ML401/ML402/ML403 Evaluation Platform - User Guide, UG080 (v2.3)*. Xilinx Inc. 2006.

[86] Open Core Protocol International Partnership (OCP-IP). *Open Core Protocol Specification 2.2*. Tech. rep. Beaverton, USA, 2006.

[87] *OPB External Memory Controller (OPB EMC) (2.00a)*. Xilinx Inc. 2006.

[88] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels". In: *IEEE Trans. Comput.* 55.2 (Feb. 2006), pp. 99–112. ISSN: 0018-9340. DOI: 10.1109/TC.2006.16. URL: http://dx.doi.org/10.1109/TC.2006.16.

[89] *PLB External Memory Controller (PLB EMC) (2.00a)*. Xilinx Inc. 2006.

[90] Gunar Schirner and Rainer Dömer. "Fast and Accurate Transaction Level Models Using Result Oriented Modeling". In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '06. San Jose, California: ACM, 2006, pp. 363–368. ISBN: 1-59593-389-1. DOI: 10.1145/1233501.1233574. URL: http://doi.acm.org/10.1145/1233501.1233574.

[91] *Virtex-4 Family Overview, DS112 (v1.5)*. Xilinx Inc. 2006.

[92] Ryan J. Wisnesky. *The Inheritance Anomaly Revisited*. http://wisnesky.net/anomaly.pdf. 2006.

[93] A. Rose, S. Swan, J. Pierce and J.-M. Fernandez. *Transaction Level Modeling in SystemC*. Whitepaper. OSCI TLM Working Group, 2005.

[94] *Block RAM (BRAM) Block (v1.00a), DS444*. Xilinx Inc. 2005.

[95] *Embedded System Tools Reference Manual - Embedded Development Kit EDK 7.1i, UG111 (v4.0)*. Xilinx Inc. 2005.

[96] Cornelia Grabbe, Frank Oppenheimer, and Thorsten Schubert. *Requirements on Hardware/Software Communication Design based on Abstract Communication Models*. ICODES Deliverable D2 (public). OFFIS Institute for Information Technology, 2005.

[97] Eike Grimpe. "Performance Optimising Hardware Synthesis of Shared Objects". PhD thesis. Universität Oldenburg, 2005.

[98] *IBM PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs*. IBM Corp. 2005.

[99] Axel Jantsch. "Models of Embedded Computation". In: *EMBEDDED SYSTEMS HANDBOOK*. CRC Press, 2005.

[100] *Joint Strike FighterAir Vehicle C++ Coding Standards for the System Development and Demonstration Program*. Lockheed Martin Corporation. 2005.

[101] *LMB BRAM Interface Controller (v1.00b), DS452*. Xilinx Inc. 2005.

[102] *Local Memory Bus (LMB) v1.0 (v1.00a), DS445*. Xilinx Inc. 2005.

[103] *MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 8.1i, UG081 (v5.3)*. Xilinx Inc. 2005.

[104] *Microprocessor Debug Module (MDM) (v2.00a), DS450*. Xilinx Inc. 2005.

[105] *On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.10c), DS401*. Xilinx Inc. 2005.

[106] *OPB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller (v1.10a), DS424*. Xilinx Inc. 2005.

[107] *OPB Interrupt Controller (v1.00c), DS473*. Xilinx Inc. 2005.

[108] *OPB IPIF (v3.01c)*. Xilinx Inc. 2005.

[109] *OPB Timer/Counter (v1.00b), DS465*. Xilinx Inc. 2005.

[110] *OPB UART Lite (v1.00b), DS422*. Xilinx Inc. 2005.

[111] Frank Oppenheimer. "OOCOSIM - An Object-Oriented Co-Design Method for Embedded HW/SW Systems". eng. PhD thesis. Uhlhornsweg 49-55, 26129 Oldenburg: Universität Oldenburg, 2005.

[112] *PLB IPIF (v2.02a).* Xilinx Inc. 2005.

[113] *Processor IP Reference Guide - OPB Usage in Xilinx FPGAs.* Xilinx Inc. 2005.

[114] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. *Transaction Level Modeling in SystemC.* Tech. rep. Open SystemC Initiative (OSCI) TLM Working Group, 2005.

[115] Dongwan Shin and Daniel D. Gajski. *Interface Synthesis from Protocol Specification.* Tech. rep. Technical Report CECS-02-13, Department of Information and Computer Science University of California, Irvine, 2005.

[116] Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. “Automatic Generation of Communication Architectures”. In: *From Specification to Embedded Systems Application.* Springer Science+Business Media, New York, NY, ISBN 0-387-27557-6, 2005.

[117] STMicroelectronics. *TAC: Transaction Accurate Communication.* http://www.greensocs.com/TACPackage. 2005.

[118] *Xilinx ISE 7 Software Manuals and Help - PDF Collection.* Xilinx Inc. 2005.

[119] Joachim K. Anlauf and Philipp A. Hartmann. “On Actors and Objects - OOP in System Level Design”. In: *FDL.* ECSI, 2004, pp. 392–404.

[120] Gerd Behrmann, Alexandre David, and Kim G. Larsen. “A Tutorial on Uppaal”. In: *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004.* Ed. by Marco Bernardo and Flavio Corradini. LNCS 3185. Springer–Verlag, 2004, pp. 200–236.

[121] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia, and Francesco Papariello. “OCCN: A NoC Modeling Framework for Design Exploration”. In: *J. Syst. Archit.* 50.2-3 (Feb. 2004), pp. 129–163. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2003.07.002. URL: http://dx.doi.org/10.1016/j.sysarc.2003.07.002.

[122] A. Jantsch. *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation.* Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, 2004. ISBN: 9781558609259. URL: http://books.google.de/books?id=hgYhEhzI72IC.

[123] ARM Ltd. *AMBA 3 AXI Protocol v1.0 Specification.* Datasheet. 2004.

[124] *MISRA-C: 2004 - Guidelines for the use of the C language in critical systems.* MIRA Limited. 2004.

[125] *OPB Block RAM (BRAM) Interface Controller, DS468.* Xilinx Inc. 2004.

[126] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. “Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration”. In: *Proceedings of the 41st Annual Design Automation Conference.* DAC ’04. San Diego, CA, USA: ACM, 2004, pp. 113–118. ISBN: 1-58113-828-8. DOI: 10.1145/996566.996603. URL: http://doi.acm.org/10.1145/996566.996603.

[127] J.M. Paul and D.E. Thomas. “Models of computation for systems-on-chip”. In: *Multiprocessor Systems-on-Chip.* Ed. by A. Jerraya and W. Wolf. Morgan Kaufman Publishers, 2004. Chap. 15.

[128] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. “Object-oriented modeling and synthesis of SystemC specifications”. In: *ASP-DAC ’04: Proceedings of the 2004 conference on Asia South Pacific design automation.* Yokohama, Japan: IEEE Press, 2004, pp. 238–243. ISBN: 0-7803-8175-0.

[129] Samar Abdi, Dongwan Shin, and Daniel Gajski. “Automatic communication refinement for system level design”. In: *DAC ’03: Proceedings of the 40th conference on Design automation.* Anaheim, CA, USA: ACM, 2003, pp. 300–305. ISBN: 1-58113-688-9. DOI: http://doi.acm.org/10.1145/775832.775911.

[130] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. "Metropolis: an integrated electronic system design environment". In: *Computer* 36.4 (2003), pp. 45–52. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1193228.

[131] Lukai Cai and Daniel Gajski. "Transaction level modeling: an overview". In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* Newport Beach, CA, USA: ACM, 2003, pp. 19–24. ISBN: 1-58113-742-7. DOI: http://doi.acm.org/10.1145/944645.944651.

[132] Lukai Cai, Shireesh Verma, and Daniel Gajski. *Comparision of SpecC and SystemC Languages for System Design.* Tech. rep. CA, USA: CECS, University of California, Irvine, 2003.

[133] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. "IPSIM: SystemC 3.0 enhancements for communication refinement". In: *Design, Automation and Test in Europe Conference and Exhibition, 2003.* 2003, 106–111 suppl. DOI: 10.1109/DATE.2003.1186680.

[134] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. "Taming heterogeneity - the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (2003), pp. 127 –144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[135] Eike Grimpe and Frank Oppenheimer. "Extending the SystemC Synthesis Subset by Object Oriented Features". In: *CODES+ISSS 2003.* 2003.

[136] Axel Jantsch and Hannu Tenhunen. "Will networks on chip close the productivity gap?" In: *Networks on chip.* Ed. by Axel Jantsch and Hannu Tenhunen. Hingham, MA, USA: Kluwer Academic Publishers, 2003, pp. 3–18. ISBN: 1-4020-7392-5. URL: http://dl.acm.org/citation.cfm?id=903951.903953.

[137] Tim Kogel, Malte Doerper, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Serge Goossens. "A Modular Simulation Framework for Architectural Exploration of On-chip Interconnection Networks". In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis.* CODES+ISSS '03. Newport Beach, CA, USA: ACM, 2003, pp. 7–12. ISBN: 1-58113-742-7. DOI: 10.1145/944645.944648. URL: http://doi.acm.org/10.1145/944645.944648.

[138] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. "Actor-Oriented Design Of Embedded Hardware And Software Systems". In: *Journal of Circuits, Systems, and Computers* 12 (2003), pp. 231–260.

[139] Henrik Theiling. "Control Flow Graphs for Real-Time System Analysis". PhD thesis. Universität des Saarlandes, 2003.

[140] *Designing Custom OPB Slave Peripherals for MicroBlaze.* Xilinx Inc. 2002.

[141] A. Gerstlauer and D. Gajski. *System-Level Abstraction Semantics.* Tech. rep. CA, USA: CECS, University of California, Irvine, 2002.

[142] Eike Grimpe, Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch, and Frank Oppenheimer. "SystemC Object-Oriented Extensions and Synthesis Features". In: *Forum on Design Languages FDL'02.* 2002.

[143] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. "A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach". In: *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS.* London, UK, UK: Springer-Verlag, 2002, pp. 18–37. ISBN: 3-540-43322-8. URL: http://dl.acm.org/citation.cfm?id=646466.691571.

[144] Majid Rabbani and Rajan Joshi. "An overview of the JPEG2000 still image compression standard". englisch. In: *Signal Processing: Image Communication* 17.1 (2002).

[145] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen. “Overview of bus-based system-on-chip interconnections”. In: *IEEE International Symposium on Circuits and Systems, 2002. ISCAS 2002.* Vol. 2. 2002, II–372–II–375 vol.2. DOI: 10.1109/ISCAS.2002.1011002.

[146] Robert Siegmund and Dietmar Müller. “A novel synthesis technique for communication controller hardware from declarative data communication protocol specifications”. In: *DAC '02: Proceedings of the 39th conference on Design automation.* New Orleans, Louisiana, USA: ACM, 2002, pp. 602–607. ISBN: 1-58113-461-4. DOI: http://doi.acm.org/10.1145/513918.514071.

[147] Robert Siegmund and Dietmar Müller. “Automatic Synthesis of Communication Controller Hardware from Protocol Specifications”. In: *IEEE Design and Test of Computers* 19.4 (2002), pp. 84–95. ISSN: 0740-7475. DOI: http://doi.ieeecomputersociety.org/10.1109/MDT.2002.1018137.

[148] J. Banks. *Discrete-event system simulation.* Prentice-Hall international series in industrial and systems engineering. Prentice Hall, 2001. ISBN: 9780130887023. URL: http://books.google.de/books?id=NV1RAAAAMAAJ.

[149] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. “Graphviz - Open Source Graph Drawing Tools”. In: *Graph Drawing'01.* 2001, pp. 483–484.

[150] Daniel D. Gajski, Rainer Domer, Junyu Peng, and Andreas Gerstlauer. *System Design: A Practical Guide with Specc.* Norwell, MA, USA: Kluwer Academic Publishers, 2001. ISBN: 0792373871.

[151] *On-Chip Peripheral Bus Architecture Specifications, SA-14-2528-02.* IBM Inc. 2001.

[152] Wolfram Putzke-Röming. “Durchgängiges Kommunikationsdesign für den strukturalen, objektorientierten Hardwareentwurf.” In: *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2001).* 2001, pp. 39–48.

[153] Wolfram Putzke-Röming. “Durchgängiges Kommunikationsdesign für den strukturalen, objektorientierten Hardware-Entwurf”. PhD thesis. Universität Oldenburg, 2001.

[154] Carsten Schulz-key, Tommy Kuhn, and Wolfgang Rosenstiel. “A Framework for System-Level Partitioning of Object-Oriented Specifications”. In: *In Proceedings of the tenth workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2001.* 2001.

[155] R. Siegmund and D. Müller. “SystemCSV - an Extension of SystemC for Mixed Multi-level Communication Modeling and Interface-based System Design”. In: *Proceedings of the Conference on Design, Automation and Test in Europe.* DATE '01. Munich, Germany: IEEE Press, 2001, pp. 26–33. ISBN: 0-7695-0993-2. URL: http://dl.acm.org/citation.cfm?id=367072.367080.

[156] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. “FunState-an internal design representation for codesign”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 9.4 (2001), pp. 524–544. ISSN: 1063-8210. DOI: 10.1109/92.931229.

[157] Kjetil Svarstad, Nezih Ben-fredj, Gabriela Nicolescu, and Ahmed A. Jerraya. “A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems”. In: *Proc. of Asia South Pacific Design Automation Conference.* 2001.

[158] Hugo A. Andrade and Margarida F. Jacome. “The Common Hardware and Software Object Model: CHSOM”. In: *PDPTA.* Ed. by Hamid R. Arabnia. CSREA Press, 2000.

[159] Prashant Arora and Rajesh K. Gupta. “Design and Implementation of a Hierarchical Exception Handling Extension to SystemC”. In: *CASES.* 2000, pp. 80–84.

[160] Daniel Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[161] Michael Philippsen. “A survey of concurrent object-oriented languages”. In: *Concurrency: Practice and Experience* 12.10 (2000), pp. 917–980. ISSN: 1096-9128. DOI: 10.1002/1096-9128(20000825)12:10<917::AID-CPE517>3.0.CO;2-F. URL: http://dx.doi.org/10.1002/1096-9128(20000825)12:10<917::AID-CPE517>3.0.CO;2-F.

[162] Martin Radetzki. “Synthesis of Digital Circuits from Object-Oriented Specifications”. PhD thesis. Carl v. Ossietzky Universität Oldenburg, 2000.

[163] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Chichester, UK: Wiley, 2000. ISBN: 978-0-471-60695-6.

[164] Philips Semiconductors. *The I2C-bus specification*. 2000.

[165] Hanyu Yin, Haito Du, Tzu-Chia Lee, and Daniel Gajski. *Design of a JPEG Encoder using SpecC Methodology*. Tech. rep. CA, USA: CECS, University of California, Irvine, 2000.

[166] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siskaand L. Sunand S. Zhao, and D. Gajski. *Design of a JPEG Encoding System*. Tech. rep. CA, USA: CECS, University of California, Irvine, 1999.

[167] ARM Ltd. *AMBA Specification (Rev 2.0)*. Datasheet. 1999.

[168] Kassem Saleh, Robert L. Probert, and Hassib Khanafer. “The distributed object computing paradigm: concepts and applications”. In: *Journal of Systems and Software* 47.2-3 (1999), pp. 125–131.

[169] A. Girault, B. Lee, and E. A. Lee. *Hierarchical Finite State Machines with Multiple Concurrency Models*. Tech. rep. UCB/ERL M97/57. Berkeley, CA 94720: Electronics Research Laboratory, 1998.

[170] E.A. Lee and A. Sangiovanni-Vincentelli. “A framework for comparing models of computation”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 17.12 (1998), pp. 1217 –1229. ISSN: 0278-0070. DOI: 10.1109/43.736561.

[171] Jakob Nielsen. *Nielsen’s Law of Internet Bandwidth*. http://www.nngroup.com/articles/nielsens-law-of-internet-bandwidth/. 1998. URL: http://www.nngroup.com/articles/nielsens-law-of-internet-bandwidth/.

[172] S. Nilsson and G. Karlsson. “Fast address look-up for internet routers”. In: *Proceedings of the IFIP TC6/WG6.2 Fourth International Conference on Broadband Communications: The future of telecommunications*. BC ’98. London, UK, UK: Chapman & Hall, Ltd., 1998, pp. 11–22. ISBN: 0-412-84410-9. URL: http://dl.acm.org/citation.cfm?id=646953.713338.

[173] J.E. Savage. *Models of computation: exploring the power of computing*. Addison Wesley, 1998. ISBN: 9780201895391. URL: http://books.google.de/books?id=j2kZAQAAIAAJ.

[174] R.G. Taylor. *Models of computation and formal language*. Oxford University Press, New York, 1998.

[175] Kim G. Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a Nutshell”. In: *Int. Journal on Software Tools for Technology Transfer* 1.1–2 (Oct. 1997), pp. 134–152.

[176] Frank Manola. *Object Model Features Matrix*. Tech. rep. X3H7-93-007v12b. National Committee for Information Technology Standards Technical Committee H7, May 1997.

[177] James A. Rowson and Alberto Sangiovanni-Vincentelli. “Interface-based design”. In: *DAC ’97: Proceedings of the 34th annual conference on Design automation*. Anaheim, California, United States: ACM, 1997, pp. 178–183. ISBN: 0-89791-920-3. DOI: http://doi.acm.org/10.1145/266021.266060.

[178] K. Salomonsen. “Design and Implementation of an MPEG/Audio Layer III Bitstream Processor”. MA thesis. Aalborg University, Denmark, 1997.

[179] Frank Vahid and Linus Tauro. “An Object-Oriented Communication Library for Hardware-Software CoDesign”. In: *Proceedings of the 5th International Workshop on Hardware/Software Co-Design*. CODES ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 81–. ISBN: 0-8186-7895-X. URL: http://dl.acm.org/citation.cfm?id=792768.793495.

[180] Wayne Wolf and Jorgen Staunstrup. *Hardware/Software CO-Design: Principles and Practice*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 0792380134.

[181] J. Zhu, R. Dömer, and D. Gajski. “Syntax and Semantics of the SpecC language”. In: *Proceedings of the SASIMI 1997*. 1997.

[182] R. Greg Lavender and Douglas C. Schmidt. “Pattern Languages of Program Design 2”. In: ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Active Object: An Object Behavioral Pattern for Concurrent Programming, pp. 483–499. ISBN: 0-201-895277. URL: http://dl.acm.org/citation.cfm?id=231958.232967.

[183] A. J. Wellings and A. Burns. “Implementing Atomic Actions in Ada 95”. In: *IEEE Transactions on Software Engineering* 23 (1996), pp. 107–123.

[184] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. “Cyclo-static data flow”. In: *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. Vol. 5. 1995, 3255–3258 vol.5. DOI: 10.1109/ICASSP.1995.479579.

[185] F. Vahid, S. Narayan, and D. D. Gajski. “SpecCharts: A VHDL Front-End for Embedded Systems”. In: *IEEE Transactions on Computer Aided Design* 14.6 (1995), pp. 694–706.

[186] Rajeev Alur and David L. Dill. “A Theory of Timed Automata”. In: *Theoretical Computer Science* 126 (1994), pp. 183–235.

[187] M. von der Beeck. “A Comparison of StateCharts Variants”. In: *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*. Berlin: Springer-Verlag, 1994, pp. 128–148.

[188] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN: 0-13-150731-1.

[189] Sanjiv Narayan and Daniel D. Gajski. “Protocol generation for communication channels”. In: *DAC ’94: Proceedings of the 31st annual conference on Design automation*. San Diego, California, United States: ACM, 1994, pp. 547–551. ISBN: 0-89791-653-0. DOI: http://doi.acm.org/10.1145/196244.196530.

[190] C. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Homewood IL: Irwin, 1993.

[191] R. Alur, C. Courcoubetis, and D. Dill. “Model-checking for real-time systems”. In: *Logic in Computer Science, 1990. LICS ’90, Proceedings., Fifth Annual IEEE Symposium on e*. 1990, pp. 414 –425. DOI: 10.1109/LICS.1990.113766.

[192] Gaetano Borriello. “A New Interface Specification Methodology and its Application to Transducer Synthesis”. PhD thesis. EECS Department, University of California, Berkeley, 1988. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/5639.html.

[193] D. D. Gajski. *Silicon Compilation*. Addison-Wesley, 1988.

[194] D. Harel. “StateCharts: a Visual Formalism for Complex Systems”. In: *Science of Programming* 8 (1987).

[195] E.A. Lee and D.G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[196] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[197] D. D. Gajski and R.H. Kuhn. “Guest Editor’s Introduction: New VLSI Tools”. In: *IEEE Computer* (1983).

[198] C. A. R. Hoare. "Communicating Sequential Processes (Reprint)". In: *Communications of the Association for Computing* 26.1 (1983), pp. 100–106. ISSN: 0001-0782 (print), 1557-7317 (electronic).

[199] C. A. R. Hoare. "Monitors: an operating system structuring concept". In: *Commun. ACM* 17.10 (1974), pp. 549–557. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/355620.361161.

[200] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming." In: *IFIP Congress*. 1974, pp. 471–475. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Kahn74.

[201] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[202] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. ISBN: 9783211811061. URL: http://books.google.de/books?id=DK-EAAAAIAAJ.

[203] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (1965), pp. 114–117. URL: http://www.intel.com/technology/mooreslaw/index.htm.

[204] C.E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1949.232969.

[205] Claude Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27 (1948), pp. 379–423, 623–656. URL: http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

[206] *Application Note 105: //yx channel_transform Channel to Channel Assignment Guidelines*. www.yxi.com/applications/105-chan2chan.pdf. Y Explorations. URL: www.yxi.com/applications/105-chan2chan.pdf.

[207] *C++ Portability Guide, Version 0.8*. http://www.mozilla.org/hacking/portable-cpp.html. URL: http://www.mozilla.org/hacking/portable-cpp.html.

[208] *Cadence - C-to-Silicon Compiler*. www.cadence.com. Cadence. URL: www.cadence.com/products/sd/silicon_compiler/.

[209] *Catapult*. calypto.com/en/products/catapult/catapult_overview. Calypto. URL: calypto.com/en/products/catapult/catapult_overview.

[210] *eXCite*. www.yxi.com/products.php. Y Explorations. URL: www.yxi.com/products.php.

[211] *Forte Design Systems - Cynthesizer*. www.forteds.com. Forte Design Systems. URL: www.forteds.com.

[212] Gigascale Systems Research Center (GSRC). *Core design technology for complex heterogeneous systems*. http://www.gigascale.org/theme/core/.

[213] *Handel-C*. www.mentor.com/products/fpga/handel-c/. Mentor Graphics. URL: www.mentor.com/products/fpga/handel-c/.

[214] *Handel-C Language Reference Manual*. www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf. Mentor Graphics. URL: www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf.

[215] *Haskell general purpose, purely functional programming language homepage*. http://www.haskell.org. URL: http://www.haskell.org.

[216] *Homepage of CoSynth GmbH & Co. KG*. https://www.cosynth.com. URL: https://www.cosynth.com.

[217] *Homepage of Edison Design Group (EDG)*. http://www.edg.com. URL: http://www.edg.com.

[218] *Homepage of Synplicity*. www.synplicity.com/. URL: www.synplicity.com/.

[219] *Homepage of the Accellera Systems Initiative*. http://www.accellera.org. URL: http://www.accellera.org.

[220] *Homepage of the ANDRES Project*. http://andres.offis.de. URL: http://andres.offis.de.

[221] *Homepage of the GNU Compiler Collection for the Xilinx MicroBlaze Soft Processor Core (mb-gcc)*. http://www.xilinx.com/guest_resources/gnu/index.htm. URL: http://www.xilinx.com/guest_resources/gnu/index.htm.

[222] *Homepage of the GNU Compiler Collection (gcc) Project*. http://gcc.gnu.org/. URL: http://gcc.gnu.org/.

[223] *Homepage of the ICODES Project*. http://icodes.offis.de. URL: http://icodes.offis.de.

[224] *Homepage of the ODETTE Project*. http://odette.offis.de. URL: http://odette.offis.de.

[225] *Homepage of the* OSSS *and* Fossy *Project*. http://www.system-synthesis.org. URL: http://www.system-synthesis.org.

[226] *Homepage of the PolyDyn Project*. http://www.offis.de/struktur/projekte/polydyn.html. URL: http://www.offis.de/struktur/projekte/polydyn.html.

[227] Underbit Technologies Inc. *MAD: MPEG Audio Decoder*. http://www.underbit.com/products/mad/.

[228] *MathWorks Hompage*. http://www.mathworks.de/. URL: http://www.mathworks.de/.

[229] *SpecC Hompage at the Center for Embedded Computer Systems (CECS)*. http://www.cecs.uci.edu/~specc/. URL: http://www.cecs.uci.edu/~specc/.

[230] *SystemCrafter*. www . systemcrafter . com. SystemCrafter Ltd. URL: www . systemcrafter.com.

[231] *SystemCrafter User Manual – Version 3.0.0*. www.systemcrafter.com/downloads/UserManual.pdf. SystemCrafter Ltd. URL: www.systemcrafter.com/downloads/UserManual.pdf.

[232] Xilinx *website*. http://www.xilinx.com. URL: http://www.xilinx.com/support/software_manuals.htm.

[233] *UPPAAL Website*. http://www.uppaal.org/. URL: http://www.uppaal.org/.

[234] *Virtex^{TM}-4 Multi-Platform FPGA*. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/. Xilinx Inc.

[235] *Vivado Design Suite*. www.xilinx.com/products/design-tools/vivado/. Xilinx. URL: www.xilinx.com/products/design-tools/vivado/.

[236] *Website of the IEEE P1076 Study Group - VHDL Analysis and Standardization Group (VASG)*. http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. URL: http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome.

[237] *Website of the IEEE P1364-2005 Group - Verilog Hardware Description Language*. http://www.verilog.com/IEEEVerilog.html. URL: http://www.verilog.com/IEEEVerilog.html.

[238] *Xilinx ISE Software Manuals Homepage*. http://www.xilinx.com/support/software_manuals.htm. URL: http://www.xilinx.com/support/software_manuals.htm.

[239] *Xilinx Platform Studio and the Embedded Development Kit (EDK) Documentation Homepage*. http://www.xilinx.com/ise/embedded/edk_docs.htm. URL: http://www.xilinx.com/ise/embedded/edk_docs.htm.

Curriculum Vitæ

Contact

Name:	Kim Nico Grüttner
Address:	Zum Sonnentau 2 D-27777 Ganderkesee Germany
E-Mail:	kim.gruettner@gmx.de
Date of birth:	6 March 1979
Place of birth:	Delmenhorst, Germany
Family status:	Married, 1 child
Nationality:	German

Languages

German (native)

English (fluent)

French (basic)

Vita

June 1998	Completion of general qualification for university entrance at "Gymnasium Ganderkesee"
August 1999	Completion of civilian service at residential care home for the elderly "Ev. luth. Wichernstift e.V." in Ganderkesee
October 1999	Beginning of computer science studies at Carl von Ossietzky Universität Oldenburg
March 2002	Intermediate examination for a diploma
April 2005	Graduation with a degree (Diploma) in computer science (with honors). Title of diploma thesis: "*Einfluss der durch Hardware-/Software-Partitionierung hervorgerufenen Kommunikation auf Leistungsdaten einer Schaltung*" (Influence of the communication, caused by hardware/software partitioning, on the performance of an integrated circuit)
May 2005	Member of research staff at OFFIS – Institute for Information Technology, Oldenburg

Oct. 2008 Beginning of PhD thesis at Carl von Ossietzky Universität Oldenburg

since Oct. 2008 Leader/Manager of the "Hardware/Software Design Methodology Group" at OFFIS – Institute for Information Technology, Oldenburg `http://www.offis.de/en/r_d_divisions/transportation/groups/hardware_software_design_methodology.html`

Nov. 2010 Internal colloquium for PhD thesis at Carl von Ossietzky Universität Oldenburg, see `http://www.informatik.uni-oldenburg.de/download/aktuell/Kolloquien/22NOV10_Gruettner.pdf`

March 2015 Defence of PhD (Dr. rer. nat.) thesis "*Application Mapping and Communication Synthesis for Object-Oriented Platform-Based Design*" (summa cum laude)

April 2015 Appointment as "Principal Scientist" at OFFIS – Institute for Information Technology, Oldenburg

Research activities

Mar. 2005 – Sept. 2007 Research and development activities in the European research project "*ICODES – Interface and Communication based Design of Embedded Systems*"
Main activities: Definition of the OSSS design methodology with a main focus on the virtual target architecture. Implementation, test and documentation of the OSSS simulation library (OSSS Shared Objects, OSSS-Channels and OSSS Remote Method Invocation (RMI)), Evaluation of the simulation framework with industrial use-cases.
More information at `http://www.system-synthesis.org`

Sept. 2006 – Jun. 2009 Research and development activities in the European research project "*ANDRES - Analysis and Design of run-time Reconfigurable, heterogeneous Systems*"
Main activities: Integration of Reconfigurable Objects into the OSSS methodology, Development of a concept for the integration of Reconfigurable Objects with the OSSS Virtual Target Architecture Layer and RMI. Implementation of RTL IP integration support for the synthesis tool FOSSY.
Support during the planning and preparation of a spin-off: CoSynth
(see `http://www.cosynth.com`)

Dec. 2009 – Mar. 2013 Coordination of the European integrated research project "*COMPLEX – COdesing and power Management in PLatform-based design space Exploration*" with 14 European partners
(see `http://complex.offis.de`)

since Oct. 2013 Coordination of the European integrated research project "*CONTREX – Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties*" with 15 European partners
(see `http://contrex.offis.de`)

Teaching activities

Oct. 2009 – Feb. 2010 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg
(see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Oct. 2009 – Dec. 2010 Co-Supervision of the student project group "*ViDAs – Virtual Driver Assistance*" of the divisions "*Safety Critical Embedded Systems*" and "*Embedded Hardware/Software Systems*" at the department for computer science at Carl von Ossietzky Universität Oldenburg
(see `http://vidas.informatik.uni-oldenburg.de/`)

Apr. 2011 – Sept. 2011 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg (see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Apr. 2012 – Sept. 2012 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg (see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Apr. 2013 – Sept. 2013 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg (see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Apr. 2014 – Sept. 2014 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg (see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Apr. 2015 – Sept. 2015 Main responsible for the module/lecture "*System-Level Design*" in the master program "*Embedded Systems & Micro-Robotics*" at the department for computer science at Carl von Ossietzky Universität Oldenburg (see `http://ehs.informatik.uni-oldenburg.de/42687.html`)

Other activities

2014 Member of the MultiPARTES (Multi-cores Partitioning for Trusted Embedded Systems) EU-Project Advisory Board (see `http://www.multipartes.eu/advisors.html`)

Membership in organizations

IEEE Institute of Electrical and Electronics Engineers Member (`http://www.ieee.org`)

HiPEAC European Network of Excellence on High Performance and Embedded Architecture and Compilation (Affiliate Member) (`http://www.hipeac.net`)

ECSI Electronic Chips & Systems design Initiative Member (`http://www.ecsi.org/`)

Membership in Program Committees

2011 ESLsyn 2011: The 2011 Electronic System Level Synthesis Conference, June 5-6, 2011 San Diego, California, USA (`http://www.ecsi.org/eslsyn2011`)

SORT 2011: 2nd IEEE Workshop on Self-Organizing Real-Time Systems (Satellite Workshop of 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing)

2012 ESLsyn 2012: The 2012 Electronic System Level Synthesis (ESLSyn) Conference, (co-located with 49th ACM/EDAC/IEEE Design Automation Conference (DAC)) June 2-3, 2012 San Francisco, California, USA (`http://www.ecsi.org/eslsyn2012`)

QVVP 2012: Quo Vadis, Virtual Platforms? Challenges and Solutions for Today and Tomorrow, Workshop at Design Automation & Test in Europe (DATE) 2012 (`http://qvvp12.offis.de`)

SORT 2012: 3rd IEEE Workshop on Self-Organizing Real-Time Systems (Satellite Workshop of 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing)

2013

ESLsyn 2013: The 2013 Electronic System Level Synthesis (ESLSyn) Conference, (co-located with 50th ACM/EDAC/IEEE Design Automation Conference (DAC)) May 31-June 1, 2013 Austin, Texas, USA
(`http://www.ecsi.org/eslsyn2013`)

IESS 2013: International Embedded Systems Symposium, 17.-19.06.2013 Paderborn, Germany
(`http://www.iess.org/`)

SORT 2013: 4th IEEE Workshop on Self-Organizing Real-Time Systems (Satellite Workshop of 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing)

2014

DAC Workshop on System to Silicon Performance Modeling and Analysis, 51th ACM/EDAC/IEEE Design Automation Conference (DAC) June 1-5, 2014 San Francisco, CA, USA
(`http://www2.dac.com/events/eventdetails.aspx?id=170-6-W`)

ESLsyn 2014: The 2014 Electronic System Level Synthesis (ESLSyn) Conference, (co-located with 51th ACM/EDAC/IEEE Design Automation Conference (DAC)) June 1-5, 2014 San Francisco, CA, USA
(`http://www.ecsi.org/eslsyn2014`)

MCSDIA 2014: Mixed-Criticality System Design, Implementation and Analysis, Special Session at 17th Euromicro Conference on Digital System Design (DSD) 27-29 August 2014, Verona, Italy
(`http://esd.scienze.univr.it/dsd-seaa-2014/?page_id=361`)
(`http://www.euromicro.org/dsd/`)

SIES 2014: 9th IEEE International Symposium on Industrial Embedded Systems
June 18-20, 2014, Pisa, Italy
(`http://retis.sssup.it/sies2014/`)

SORT 2014: 5th IEEE Workshop on Self-Organizing Real-Time Systems (Satellite Workshop of 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing)
(`http://www.es.cs.uni-frankfurt.de/?id=sort2014`)

2015

DATE 2015: Design, Automation & Test in Europe
Topic D2 - "System Design, High-Level Synthesis and Optimization"
March 9-13, 2015, Grenoble, France
(`http://www.date-conference.com/`)

1st International Workshop on Investigating Dataflow in Embedded computing Architecture (IDEA), in conjunction with with HiPEAC 2015 conference January 19-21, 2015, Amsterdam, The Netherlands
(`http://caes.ewi.utwente.nl/idea2015`)
(`http://www.hipeac.net/2015/amsterdam`)

3rd International workshop on the "Integration of mixed-criticality subsystems on multi-core and manycore processors", in conjunction with with HiPEAC 2015 conference, January 19-21, 2015, Amsterdam, The Netherlands
(`http://www.hipeac.net/2015/amsterdam`)

SIES 2015: 10th IEEE International Symposium on Industrial Embedded Systems, June 8-10, 2015, Sigen, Germany
(`http://www.sies2015.com/`)

SORT 2015: 6th IEEE Workshop on Self-Organizing Real-Time Systems (Satellite Workshop of 17th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing) (`http://www.es.cs.uni-frankfurt.de/?id=sort2015`)

MCSDIA 2015: Mixed-Criticality System Design, Implementation and Analysis, Special Session at 18th Euromicro Conference on Digital System Design (DSD) August 26-28, 2015, Funchal, Madeira, Portugal
(`http://paginas.fe.up.pt/~dsd-seaa-2015/dsd2015/`)
(`http://www.euromicro.org/dsd/`)

Embedded Multi-Core Systems for Mixed Criticality Applications in Dynamic and Changeable Real-Time Environments, Special Session at IEEE International Conference on Industrial Informatics (INDIN'15), July 22-24, 2015, Cambridge, UK
(`http://ww2.anglia.ac.uk/ruskin/en/home/microsites/indin_2015/special_sessions_/special_session_documents.html`)

ESLsyn 2015: The 2015 Electronic System Level Synthesis (ESLSyn) Conference, (co-located with 52th ACM/EDAC/IEEE Design Automation Conference (DAC)) June 10-11, 2015, San Francisco, CA, USA
(`http://www.ecsi.org/eslsyn`)

EUROCON 2015: 16th International Conference on Computer as a Tool September 8-11, 2015, Salamanca, Spain
(`http://eurocon2015.usal.es/`)

2016 DATE 2016: Design, Automation & Test in Europe
Topic D2 - "System Design, High-Level Synthesis and Optimization"
March 14-18, 2016, Dresden, Germany
(`http://www.date-conference.com/`)

Organization and Co-Organization of Workshops and Conferences

2011 MBMV 2011: 14. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 21.-23.02.2011, OFFIS e.V., Oldenburg
(`http://mbmv2011.offis.de/`)

2012 QVVP 2012: Quo Vadis, Virtual Platforms? Challenges and Solutions for Today and Tomorrow, Workshop at Design Automation & Test in Europe (DATE) 2012
(`http://qvvp12.offis.de`)

CODES+ISSS'12 Special Session: Synthesis of Executable Extra-Functional System-Level Models for Timing and Power Exploration, Embedded Systems Week 2012, 7th-12th October 2012, Tampere, Finland
(`http://esweek.acm.org/esweek2012/codesisss/`)

2014 DAC Workshop on System to Silicon Performance Modeling and Analysis, 51th ACM/EDAC/IEEE Design Automation Conference (DAC) June 1-5, 2014 San Francisco, CA, USA
(`http://www2.dac.com/events/eventdetails.aspx?id=170-6-W`)

MCSDIA 2014: Mixed-Criticality System Design, Implementation and Analysis, Special Session at 17th Euromicro Conference on Digital System Design (DSD) August 27-29, 2014, Verona, Italy
(`http://esd.scienze.univr.it/dsd-seaa-2014/?page_id=361`)
(`http://www.euromicro.org/dsd/`)

2015 3rd International workshop on the "Integration of mixed-criticality subsystems on multi-core and manycore processors", in conjunction with with

HiPEAC 2015 conference, January 19-21, 2015, Amsterdam, The Netherlands
(`http://www.hipeac.net/2015/amsterdam`)

DAC Workshop on System-to-Silicon Performance Modeling and Analysis, 52th ACM/EDAC/IEEE Design Automation Conference (DAC) June 07, 2015 San Francisco, CA, USA
(`http://www2.dac.com/events/eventdetails.aspx?id=182-4-W`)

MCSDIA 2015: Mixed-Criticality System Design, Implementation and Analysis, Special Session at 18th Euromicro Conference on Digital System Design (DSD) August 26-28, 2015, Funchal, Madeira, Portugal
(`http://paginas.fe.up.pt/~dsd-seaa-2015/dsd2015/`)
(`http://www.euromicro.org/dsd/`)

High Integrity Multi-Core Modelling for Future Systems (Hi-MCM), Special Session at Forum on specification & Design Languages (FDL) September 14-16, 2015, Barcelona, Spain
(`http://www.ecsi.org/fdl`)
(`http://www.ecsi.org/sites/default/files/PDF/fdl2015_cfp_specialsession_HiMCM.PDF`)

Publications and Talks

Thesis

[Th1] Kim Grüttner. “Application Mapping and Communication Synthesis for Object-Oriented Platform-Based Design”. Doktorarbeit. Carl von Ossietzky Universität Oldenburg, Fakultät II - Department für Informatik, Abteilung Eingebettete Hardware-/Software-Systeme, 2015.

[Th2] Kim Grüttner. “Einfluss der durch Hardware-/Software-Partitionierung hervorgerufenen Kommunikation auf Leistungsdaten einer Schaltung”. Diplomarbeit. Carl von Ossietzky Universität Oldenburg, Fakultät II - Department für Informatik, Abteilung Eingebettete Hardware-/Software-Systeme, 2005.

[Th3] Kim Grüttner. “Blinde Quellentrennung – Eine grundlegende Einführung”. Studienarbeit. Carl von Ossietzky Universität Oldenburg, Institut für Physik, Arbeitsgruppe Signalverarbeitu, 2004.

[Th4] Alexander Borgerding, Johannes Faber, Kim Grüttner, Thomas Heuer, Baltin Karro, Stephanie Kemper, Arne Limburg, Iris Menge, Stefan Puch, and Arno Willig. “Projektgruppe Kooperierende autonome Systeme (KautS) Endbericht”. Projektgruppenendbericht. Carl von Ossietzky Universität Oldenburg, Fakultät II - Department für Informatik, Abteilung Entwicklung korrekter Systeme, 2003.

Directly related to this dissertation

Articles

[AD1] Tim Schmidt, Kim Grüttner, Rainer Dömer, and Achim Rettberg. “A program state machine based virtual processing model in SystemC”. In: *SIGBED Review* 11.4 (2014), pp. 7–12. DOI: `10.1145/2724942.2724943`. URL: `http://doi.acm.org/10.1145/2724942.2724943`.

Book chapters

[BD1] Matthias Bücker, Kim Grüttner, Philipp A. Hartmann, and Ingo Stierand. "System Specification and Design Languages – Selected Contributions from FDL 2010". In: Springer, Jan. 2012. Chap. Mapping of Concurrent Object-Oriented Models to Extended Real-Time Task Networks. ISBN: 978–1–4614–1426–1.

Conference papers

[PD1] Tim Schmidt, Kim Grüttner, Rainer Dömer, and Achim Rettberg. "A Program State Machine Based Virtual Processing Model in SystemC". In: *Proceedings of the Embed With Linux 2014 Workshop, Lisboa, Portugal, November 13-14, 2014.* Ed. by Jalil Boukhobza, Jean-Philippe Diguet, Pierre Ficheux, José Rufino, and Frank Singhoff. Vol. 1291. CEUR Workshop Proceedings. CEUR-WS.org, 2014. URL: `http://ceur-ws.org/Vol-1291/ewili14_3.pdf`.

[PD2] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, and Achim Rettberg. "Ein generisches Treiber-Framework zur HW/SW-Kommunikation mittels OSSS-RMI". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'2011).* Feb. 2011.

[PD3] Matthias Büker, Kim Grüttner, Philipp A. Hartmann, and Ingo Stierand. "Mapping of Concurrent Object–Oriented Models to Extended Real–Time Task Networks". In: *Forum on Specification & Design Languages (FDL).* Sept. 2010.

[PD4] Kim Grüttner, Henning Kleen, Frank Oppenheimer, Achim Rettberg, and Wolfgang Nebel. "Towards a Synthesis Semantics for SystemC Channels". In: *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS).* Oct. 2010.

[PD5] Philipp A. Hartmann, Kim Grüttner, Achim Rettberg, and Ina Podolski. "Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC". In: *7th IFIP Conference on Distributed and Parallel Embedded Systems (DIPES).* Sept. 2010.

[PD6] Kim Grüttner, Frank Oppenheimer, Wolfgang Nebel, Jan Freuer, and Joachim Gerlach. "Rapid Prototyping und Synthese eines videobasierten Fahrerassistenzsystems mit C++ und SystemC(TM)". In: *10. Braunschweiger Symposium AAET 2009 – Automatisierungs- , Assistenzsysteme und eingebettete Systeme für Transportmittel.* Feb. 2009.

[PD7] Kim Grüttner and Wolfgang Nebel. "Modelling Program–State Machines in SystemC(TM)". In: *Forum on Specification and Design Languages (FDL).* Sept. 2008.

[PD8] Kim Grüttner, Frank Oppenheimer, and Wolfgang Nebel. "OSSS Methodology – System-Level Design and Synthesis of Embedded HW/SW Systems in C++". In: *First International Symposium on Applied SCiences in Bio-Mediacl and Communication Technologies (ISABEL).* Jan. 2008.

[PD9] Kim Grüttner, Frank Oppenheimer, Wolfgang Nebel, Fabien Colas-Bigey, and Anne-Marie Fouilliart. "SystemC-based Modelling, Seamless Refinement, and Synthesis of a JPEG 2000 Decoder". In: *Design, Automation, & Test in Europe (DATE) Conference.* Apr. 2008.

[PD10] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer, and Wolfgang Nebel. "Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS". In: *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI).* Oct. 2007.

[PD11] Cornelia Grabbe, Claus Brunzema, Kim Grüttner, Thorsten Schubert, and Frank Oppenheimer. "Overview of the ICODES Project". In: *Forum on Specification & Design Languages (FDL).* Sept. 2006. ISBN: 3–00–019710–9.

[PD12] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer, and Wolfgang Nebel. "Modelling and Synthesis of Communication Using OSSS-Channels". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. 9. ITG/GI/GMM Workshop. Fachgruppe 3 und 4 der RSS Kooperationsgemeinschaft Rechnergestützter Schaltungs- und Systementwurf der GI, ITG und GMM. Feb. 2006. ISBN: 3–9810287–1–6.

[PD13] Kim Grüttner, Claus Brunzema, Cornelia Grabbe, Thorsten Schubert, and Frank Oppenheimer. "OSSS-Channels: Modelling and Synthesis of Communication With SystemC". In: *Forum on Specification & Design Languages (FDL)*. Sept. 2006. ISBN: 3–00–019710–9.

Technical reports

[TD1] Cornelia Grabbe, Kim Grüttner, Thorsten Schubert, and Frank Oppenheimer. *Specification of Hardware/Software Communication Design Methodology based on Abstract Communication Models*. Tech. rep. OFFIS - Institute for Information Technology, Aug. 2005.

Manuals

[MD1] Kim Grüttner. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – The OSSS 2.2.0 Tutorial*. OFFIS - Institute for Information Technology. Sept. 2008.

[MD2] Kim Grüttner, Andreas Herrholz, Philipp A. Hartmann, Andreas Schallenberg, and Claus Brunzema. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – The OSSS 2.2.0 Manual*. OFFIS - Institute for Information Technology. Sept. 2008.

[MD3] Claus Brunzema, Cornelia Grabbe, Kim Grüttner, Philipp Andreas Hartmann, Andreas Herrholz, Henning Kleen, Frank Oppenheimer, Andreas Schallenberg, Christian Stehno, and Thorsten Schubert. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – A tutorial for OSSS 2.0*. OFFIS - Institute for Information Technology. Jan. 2007.

Related topics

Articles

[AR1] Kim Grüttner, Philipp A. Hartmann, Kai Hylla, Sven Rosinger, Wolfgang Nebel, Fernando Herrera, Eugenio Villar, Carlo Brandolese, William Fornaciari, Gianluca Palermo, Chantal Ykman-Couvreur, Davide Quaglia, Francisco Ferrero, and Raúl Valencia. "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration". In: *Microprocessors and Microsystems* 37.8, Part C (2013). Special Issue on European Projects in Embedded System Design: EPESD2012, pp. 966 –980. ISSN: 0141-9331. DOI: `http://dx.doi.org/10.1016/j.micpro.2013.09.001`. URL: `http://www.sciencedirect.com/science/article/pii/S0141933113001221`.

Book chapters

[BR1] Kim Grüttner, Kai Hylla, Sven Rosinger, and Wolfgang Nebel. "System Specification and Design Languages – Selected Contributions from FDL 2010". In: Springer, Jan. 2012. Chap. Rapid Prototyping of Complex HW/SW Systems using a Timing and Power Aware ESL Framework. ISBN: 978–1–4614–1426–1.

[BR2] Kim Grüttner, Philipp A. Hartmann, Andreas Herrholz, and Frank Oppenheimer. "Reconfigurable Computing – From FPGAs to Hardware/Software Codesign". In: Springer, Sept. 2011. Chap. ANDRES − Analysis and Design of Run-Time Reconfigurable, Heterogeneous Systems. ISBN: 978–1–4614–0060–8.

[BR3] Andreas Schallenberg, Wolfgang Nebel, Andreas Herrholz, Philipp A. Hartmann, Kim Grüttner, and Frank Oppenheimer. “Dynamically Reconfigurable Systems Architectures, Design Methods and Applications”. In: Springer, Dec. 2009. Chap. POLYDYN Object-oriented modelling and synthesis targeting dynamically reconfigurable FPGAs. ISBN: 978–90–481–3484–7.

Conference papers

[PR1] Philipp A. Hartmann, Kim Grüttner, and Wolfgang Nebel. “Advanced SystemC Tracing and Analysis Framework for Extra–Functional Properties”. In: *The 11th International Symposium on Applied Reconfigurable Computing (ARC’15)*. Apr. 2015.

[PR2] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, and Wolfgang Nebel. “A Workload Extraction Framework for Software Performance Model Generation”. In: *7th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*. Jan. 2015.

[PR3] Kim Grüttner, Philipp A. Hartmann, Tiemo Fandrey, Kai Hylla, Daniel Lorenz, Stefan Stattelmann, Björn Sander, Oliver Bringmann, Wolfgang Nebel, and Wolfgang Rosenstiel. “An ESL Timing & Power Estimation and Simulation Framework for Heterogeneous SoCs”. In: *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), Samos, Greece, July 14–17, 2014*. July 2014.

[PR4] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. “Exploiting Segregation in Bus-Based MPSoCs to Improve Scalability of Model-Checking-Based Performance Analysis for SDFAs”. In: *International Embedded Systems Symposium (IESS)*. June 2013.

[PR5] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. “Towards Performance Analysis of SDFGs Mapped to Shared–Bus Architectures Using Model–Checking”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE) 2013*. DATE ’13. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, Mar. 2013.

[PR6] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, and Wolfgang Nebel. “Ansatz zur Bewertung der HW/SW–Kommunikation in asymmetrischen Multi–Prozessor–Systemen”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV’2013)*. Universität Rostock. Mar. 2013, pp. 197–208.

[PR7] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, and Achim Rettberg. “Hierarchical Real–Time Scheduling in the Multi–Core Era – An Overview”. In: *SORT 2013 – The Fourth IEEE Workshop on Self-Organizing Real-Time Systems*. June 2013.

[PR8] Maher Fakih and Kim Grüttner. “Virtual-Platform in the Loop Simulation for Accurate Timing Analysis of Embedded Software on Multicore Platforms”. In: *ASIM STS/GMMS Workshop*. GI ASIM Fachgruppen "Simulation technischer Systeme" (STS) und "Grundlagen und Methoden in Modellbildung und Simulation" (GMMS). Feb. 2012.

[PR9] Kim Grüttner, Philipp A. Hartmann, Kai Hylla, Sven Rosinger, Wolfgang Nebel, Eugenio Herrera Fernando: Villar, Carlo Brandolese, William Fornaciari, Gianluca Palermo, Chantal Ykman-Couvreur, Davide Quaglia, Francisco Ferrero, and Raul Valencia. “COMPLEX – COdesign and power Management in PLatform-based design space EXploration”. In: *15th Euromicro Conference on Digital System Design (DSD)*. Euromicro. Sept. 2012.

[PR10] Daniel Lorenz, Kim Grüttner, Nicola Bombieri, Valerio Guarnieri, and Sara Bocchio. “From RTL IP to Functional System–Level Models with Extra-Functional Properties”. In: *CODES+ISSS’12*. Oct. 2012.

[PR11] Daniel Lorenz, Philipp A. Hartmann, Kim Grüttner, and Achim Rettberg. "Nicht-invasive Simulation des Energieverbrauchs von Hardware-Komponenten auf Systemebene mit SystemC". In: *15. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Mar. 2012.

[PR12] Daniel Lorenz, Philipp A. Hartmann, Kim Grüttner, and Wolfgang Nebel. "Non-invasive Power Simulation at System-Level with SystemC". In: *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Sept. 2012.

[PR13] Maher Ali Fakih, Frank Poppen, Kim Grüttner, and Achim Rettberg. "Simulink and Virtual Hardware Platform Co-Simulation for Accurate Timing Analysis of Embedded Control Software". In: *ASIM–Konferenz STS/GMMS 2011*. ASIM/GI-Fachgruppe. Shaker Verlag, Feb. 2011. ISBN: 978–3–8322–9872–2.

[PR14] Kim Grüttner, Philipp A. Hartmann, Philipp Reinkemeier, Frank Oppenheimer, and Wolfgang Nebel. "Challenges of Multi- and Many-Core Architectures for Electronic System-Level Design". In: *SAMOS 2011: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI)*. July 2011.

[PR15] Philipp A. Hartmann, Kim Grüttner, Philipp Ittershagen, and Achim Rettberg. "A Framework for Generic HW/SW Communication using Remote Method Invocation". In: *ESLsyn - The 2011 Electronic System Level Synthesis Conference*. ECSI. June 2011.

[PR16] Philipp A. Hartmann, Kim Grüttner, Frank Oppenheimer, and Wolfgang Nebel. "Flexible Mapping of Concurrent Object-Oriented Applications to MPSoC Platforms". In: *Map2MPSoC Workshop*. ArtistDesign NoE. June 2011.

[PR17] Frank Poppen, Roland Koppe, Axel Hahn, and Kim Grüttner. "Impact Simulation of Changes to Development Processes: An ESL Case Study". In: *Forum on specification & Design Languages (FDL)*. Sept. 2011.

[PR18] Kim Grüttner, Kai Hylla, Sven Rosinger, and Wolfgang Nebel. "Towards an ESL Framework for Timing and Power Aware Rapid Prototyping of HW/SW Systems". In: *Forum on Specification & Design Languages (FDL)*. Sept. 2010.

[PR19] Andreas Popp, Andreas Herrholz, Kim Grüttner, Yannick Le Moullec, Peter Koch, and Wolfgang Nebel. "SystemC-AMS SDF Model Synthesis for Exploration of Heterogeneous Architectures". In: *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. Apr. 2010.

[PR20] Kim Grüttner, Carsten Beth, and Wolfgang Nebel. "Kommunikationsgetriebene Hardware-/Software-Partitionierung eines Netzwerkprotokollstacks auf einer SoC-Plattform". In: *INFORMATIK 2005 – Informatik LIVE!, Band 1*. Sept. 2005. ISBN: 3–88579–396–2.

Technical reports

[TR1] Philipp Reinkemeier, Philipp Ittershagen, Ingo Stierand, Philipp A. Hartmann, Stefan Henkler, and Kim Grüttner. *Seamless Segregation for Multi-Core Systems*. Tech. rep. OFFIS e.V., Aug. 2013. URL: `http://ses.informatik.uni-oldenburg.de/download/bib/paper/OFFIS-TR2013_SegregationMultiCore_20130805.pdf`.

[TR2] J. Wenninger, M. Damm, J. Haase, J. Ou, K. Grüttner, Hartmann P. A., A. Herrholz, F. Herrera, I Sander, and J. Zhu. *Overall Modelling Framework for AHES (Adaptive Heterogeneous Embedded Systems)*. Tech. rep. OFFIS - Institute for Information Technology, Aug. 2009.

Other

Articles

[AO1] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. "State-Based Real-Time Analysis of SDF Applications on MPSoCs with Shared Communication Resources". In: *Journal of Systems Architecture* 0 (2015), pp. –. ISSN: 1383-7621. DOI: http://dx.doi.org/10.1016/j.sysarc.2015.04.005. URL: http://www.sciencedirect.com/science/article/pii/S1383762115000326.

Conference papers

[PO1] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. "State-Based Real-Time Analysis of SDF Applications on Multi-Cores". In: *1st International Workshop on Investigating Dataflow in Embedded computing Architecture (IDEA)*. Jan. 2015.

[PO2] Sören Schreiner, Kim Grüttner, Sven Rosinger, and Wolfgang Nebel. "Ein Verfahren zur Bestimmung eines Powermodells von Xilinx MicroBlaze MPSoCs zur Verwendung in Virtuellen Plattformen". In: *18. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2015)*. Mar. 2015.

[PO3] Maher Fakih, Kim Grüttner, Martin Fränzle, and Achim Rettberg. "Multicore Performance analysis of a Multi-phase Electrical Motor Controller". In: *Embedded Real Time Software and Systems Congress (ERTS2) 2014*. Feb. 2014.

[PO4] Domenik Helms, Kim Grüttner, Reef Eilers, Malte Metzdorf, Kai Hylla, Frank Poppen, and Wolfgang Nebel. "Considering Variation and Aging in a Full Chip Design Methodology at System Level". In: *Proceedings of The 2014 Electronic System Level Synthesis Conference (ESLsyn'14), May 31-Jun 01 2014, San Francisco, CA, USA*. ECSI. May 2014.

[PO5] Daniel Lorenz, Kim Grüttner, and Wolfgang Nebel. "Data- and State-Dependent Power Characterisation and Simulation of Black-Box RTL IP Components at System-Level". In: *17th Euromicro Conference on Digital Systems Design (DSD 2014)*. Aug. 2014.

[PO6] Daniel Lorenz, Vincent Ortland, and Kim Grüttner. "Trace-Based Power State Machine Modelling". In: *Forum on specification & Design Languages (FDL 2014)*. Oct. 2014.

[PO7] Gregor Nitsche, Kim Grüttner, and Wolfgang Nebel. "Towards Satisfaction Checking of Power Contracts in UPPAAL". In: *Forum on specification & Design Languages (FDL 2014)*. Oct. 2014.

[PO8] J.-H. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, K. Grüttner, Th. Kruse, C. Kuznik, H. M. Le, A. Mauderer, W. Müller, D. Müller-Gritschneder, F. Poppen, H. Post, S. Reiter, W. Rosenstiel, S. Roth, U. Schlichtmann, A. von Schwerin, B.-A. Tabacaru, and A. Viehl. "Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges". In: *Proceedings of the 51th Design Automation Conference (DAC) 2014, San Francisco, CA, USA*. June 2014.

[PO9] Sören Schreiner, Kim Grüttner, and Sven Rosinger. "Autonomous flight control meets custom payload processing: A mixed-critical avionics architecture approach for civilian UAVs". In: *Proceedings of the 5th IEEE Workshop on Self-Organizing Real-Time Systems*. June 2014.

[PO10] Salvador Trujillo, Roman Obermaisser, Kim Grüttner, Francisco J. Cazorla, and Jon Perez. "European Project Cluster on Mixed-Criticality Systems". In: *Performance, Power and Predictability of Many-Core Embedded Systems (3PMCES) Workshop*. Mar. 2014.

[PO11] Jörg Walter, Maher Fakih, and Kim Grüttner. "Hardware-Based Real-Time Simulation on the Raspberry Pi". In: *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014)*. Jan. 2014.

[PO12] Gregor Nitsche, Kim Grüttner, and Wolfgang Nebel. "Power Contracts: A Formal Way Towards Power–Closure?!" In: *23th International Workshoip on Power and Timng Modeling, Optimization and Simulation (PATMOS'13)*. IEEE, Sept. 2013.

[PO13] Matthias Sauppe, Thomas Horn, Erik Markert, Ulrich Heinkel, Daniel Lorenz, Kim Grüttner, Hans-Werner Sahm, and Klaus-Holger Otto. "A Database for the Integration of Power Data on System Level". In: *EUROCON 2013, International Conference on Computer as a Tool*. IEEE. July 2013.

[PO14] Frank Poppen and Kim Grüttner. "Co-Simulation of C-based SoC Simulators and MATLAB Simulink". In: *Simulation Workshop 2012 (SW12)*. Operational Research Society. Mar. 2012.

[PO15] Kim Grüttner, Andreas Herrholz, Ulrich Kühne, Daniel Große, Achim Rettberg, Wolfgang Nebel, and Rolf Drechsler. "Towards Dependability-aware Design of Hardware Systems using extended Program State Machines". In: *SORT 2011: 2nd IEEE Workshop on Self-Organizing Real-Time Systems*. Mar. 2011.

[PO16] Sergio Montenegro, Benjamin Vogel, Vladimir Petrovic, Gunter Schoof, Andreas Herrholz, and Kim Grüttner. "Spacecraft Area Network (SCAN) for Plug and Play of Devices". In: *Small Satellite Systems and Services - The 4S Symposium*. May 2010.

[PO17] Philipp Reinkemeier, Kim Grüttner, and Wolfgang Nebel. "Eine Fallstudie zur dynamischen Rekonfiguration von Hardware: "Pain or Gain?"" In: *10. ITG/GMM/GI–Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*. Mar. 2007.

Technical reports

[TO1] Joseph Borel (ed.) *European Design Automation Roadmap 6th Edition – Design Solutions for Europe*. Roadmap 6th edition. MEDEA+/CATRENEOffice, Mar. 2009.

Lectures, Talks and Oral Presentations

[L1] Kim Grüttner. *CONTREX – Design of embedded mixed-criticality CONTRolsystems under consideration of EXtra-functional properties, at Speakers Corner of ARTEMIS/ITEA Co-Summit 2015*. Mar. 2015.

[L2] Kim Grüttner. *Empowering mixed-critical system engineers in the dark silicon era: Towards power, temperature and aging analysis of heterogeneous MPSoCs at system-level,* ***Keynote*** *at 1st Workshop on Model-Implementation Fidelity (MiFi) at DATE'15*. Mar. 2015.

[L3] Kim Grüttner. *Towards model-based power and temperature analysis of heterogeneous MPSoCs at system-level, at Kolloquium des Graduiertenkollegs "Heterogene Bildsysteme", Friedrich-Alexander-Universität Erlangen Nürnberg, 2015*. Apr. 2015.

[L4] Kim Grüttner. *Towards power, temperature and aging analysis and estimation for SoCs at system-level, at MCS: Integration of mixed-criticality subsystems on multi-core and manycore processors, HiPEAC Conference 2015*. Jan. 2015.

[L5] Kim Grüttner. *Considering Variation and Aging in a FullChip Design Methodology at System Level, at Colloquium of Center for Embedded and Cyber-physical Systems at University of California, Irvine, USA*. June 2014.

[L6] Kim Grüttner. *Design of embedded mixed-criticality control systems under consideration of extra-functional properties, at 2nd International workshop on the Integration of mixed-criticality subsystems on multi-core and manycore processors, HiPEAC Conference 2014*. Jan. 2014.

[L7] Kim Grüttner. *Modelling, simulation and analysis of mixed-criticality systems under consideration of extra-functional properties, at 3. Kolloquium Multi-core und Funktionale Sicherheit in der Automobilindustrie, Infineon, Neubiberg*. Nov. 2014.

[L8] Kim Grüttner. *Modelling, Simulation and Analysis of Mixed-Criticality Systems Under Consideration of Extra-functional Properties, at DAC Workshop on System-to-Silicon Performance Modeling and Analysis*. June 2014.

[L9] Kim Grüttner. *CONTREX – Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties, at Cyber-Physical Systems: Uplifting Europe's innovation capacity*. European Commission Directorate General CONNECT with the support of the ARTEMIS Joint Technology Initiative and Steinbeis-Europa-Zentrum, Oct. 2013.

[L10] Kim Grüttner, Philipp A. Hartmann, and Frank Oppenheimer. *Performance and Energy Modeling and Analysis in COMPLEX Virtual Platform, at embedded world Conference 2013*. Feb. 2013.

[L11] Kim Grüttner and Frank Oppenheimer. *The COMPLEX Virtual Platform Design Approach for Performance and Energy Efficient Embedded Systems: A European Research Perspective, at Embedded SW Development on Virtual Platforms Workshop - Ready for Prime Time?, in conjunction with embedded World 2012 Conference*. Feb. 2012.

[L12] Kim Grüttner, Philipp A. Hartmann, Tiemo Fandrey, Kai Hylla, Domenik Helms, Frank Oppenheimer, Wolfgang Nebel, and Achim Rettberg. *Towards Performance and Energy Efficient Embedded System Design using Virtual Platforms, at The 2012 Electronic System Level Synthesis Conference (ESLsyn)*. June 2012.

[L13] Wolfgang Nebel, Domenik Helms, Kim Grüttner, and Frank Oppenheimer. *Über die Notwendigkeit neuer Modellierungskonzepte komplexer eingebetteter Systeme, Keynote at edaWorkshop 2012*. May 2012.

[L14] Francisco Ferrero, Kim Grüttner, Fernando Herrera, Gianluca Palermo, Bart Vanthournout, and Emmanuel Vaumorin. *Using the COMPLEX Design Flow for Space Domain Applications, at Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2011), in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2011*. Mar. 2011.

[L15] Kim Grüttner. *Challenges in SoC System Synthesis, Panel Discussion: "ESL Synthesis? Get Real", at the 2011 Electronic System Level Synthesis Conference (ESLsyn)*. June 2011.

[L16] Kim Grüttner. *The COMPLEX ESL Framework for Timing and Power Aware Rapid Prototyping of HW/SW Systems, at ET12: Early Timing and Power Information of Complex SoC Designs using Augmented Virtual Platforms, in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2011*. Mar. 2011.

[L17] Kim Grüttner, Kai Hylla, Sven Rosinger, Philipp A. Hartmann, and Wolfgang Nebel. *Enabling Timing and Power Aware Virtual Prototyping of HW/SW Systems, at Workshop on Micro Power Management for Macro Systems on Chip (uPM2SoC), in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2011*. Mar. 2011.

[L18] Philipp A. Hartmann, Maher A. Fakih, and Kim Grüttner. *Non-intrusive TLM-2.0 Transaction Observation, Interception, and Augmentation, at 24th European SystemC User's Group Meeting, in conjunction with FDL 2011*. Sept. 2011.

[L19] Philipp A. Hartmann, Philipp Ittershagen, Kim Grütter, Frank Oppenheimer, and Achim Rettberg. *A Framework for Generic HW/SW Communication using Remote Method Invocation, at Workshop on Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2011). in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2011*. Mar. 2011.

[L20] Kim Grüttner. *Application Mapping and Communication Synthesis for Object-Oriented Platform-Based Design, Internes Kolloquium der Fakultät II - Department für Informatik an der Carl von Ossietzky Universität Oldenburg*. Nov. 2010.

[L21] Kim Grüttner and Frank Oppenheimer. *ANDRES – Analysis and Design of run-time Reconfigurable, heterogeneous Systems, at W1: The European landscape of reconfigurable computing: Lessons learned, new perspectives and innovations, in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2010*. Mar. 2010.

[L22] Philipp A. Hartmann, Kim Grüttner, and Frank Oppenheimer. *Exploiting Parallel Computing Platforms with OSSS, at edaWorkshop 2010*. May 2010.

[L23] Philipp A Hartmann, Kim Grüttner, Frank Oppenheimer, and Achim Rettberg. *Exploiting Parallel Computing Platforms with OSSS, at W3: Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications, in conjunction with Design, Automation, and Test in Europe Conference (DATE) 2010*. Mar. 2010.

[L24] Frank Oppenheimer and Kim Grüttner. *Objektorientierter Entwurf und Synthese von Hardware-/Softwaresystemen, Informatik-Kolloquium der Universität Bremen*. May 2007.

[L25] Frank Oppenheimer and Kim Grüttner. *OSSS: An Approach for Modelling, seamless Refinement, and Synthesis of HW/SW SoC, at 16. European SystemC User's Group Meeting, in conjunction with FDL 2007*. Sept. 2007.

[L26] Frank Oppenheimer and Kim Grüttner. *Objektorientierter Entwurf und Synthese von Hardware-/Softwaresystemen unter besonderer Berücksichtigung der Hardware-/Software-Kommunikation, Kolloquium des Instituts für Informatik an der TU Braunschweig*. Dec. 2006.

Supervised and co-supervised bachelor and master theses

[S1] Christian Nesemann. “Evaluierung von PMD-Sensorik für Fahrerassistenzsysteme”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2013.

[S2] Tim Schmidt. “A Program State Machine Based Virtual Processing Model in SystemC™”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2013.

[S3] Sören Schreiner. “Entwicklung eines bordeigenen Syetems zum autonomen Starten und Landen von mehrmotorigen Helikoptern”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2013.

[S4] Henning Schlender. “Untersuchung des Potentials von Controller- und Compiler-spezifischen Optimierungen bei der automatischen Generierung von C-Code”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2012.

[S5] Maher Ali Fakih. “Timing Validation of Functional Models on Virtual Platforms”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2011.

[S6] Fabian Meyen. “Entwicklung einer eingebetteten Diagnose Software für ErgoControl2 Steuergeräte”. Master Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2011.

[S7] Ina Podolski. “Modelling, Implementing and Validating a Kalman Filter with an Artificial Neuronal Network for Embedded Systems”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2011.

[S8] Björn Grönewold. “Entwicklung einer API zur Anbindung einer GUI an das Echtzeitbetriebssystem RODOS”. Diploma Thesis (Diplomarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2010.

[S9] Sören Schreiner. “Modellbasierter Entwurf, Validierung und Verifizierung der sicherheitskritischen Software eines Quadrokopters”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2010.

[S10] Alexander Stühring. “Entwicklung eines Target Simulation Moduls für einen Doppelkernprozessor zur Durchführung einer Prozessor-In-The-Loop Simulation mit Target Link”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2010.

[S11] Tobias Tiemerding. “Implementierung eines Displaytreibers zur CAN-Bus Diagnose auf einem externen Display”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2010.

[S12] Oliver Miller. “Entwurf und Implementierung eines Kommunikationscoprozessors zur Remote Method Invocation für die SoC-Kommunikation”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2008.

[S13] Christian Ammann. “Multiprocessors on Chip”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2006.

[S14] Kai Hylla. “Evaluierung busbasierter Kommunikationsprotokolle im SoC-Design und Entwurf eines abstrakten Interfaces”. Bachelor’s Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2006.

[S15] Gerold Mauson. “Extraktion und Visualisierung von Strukturinformationen aus SystemC-Modellen”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2006.

[S16] Philipp Reinkemeier. “Laufzeit-Rekonfigurierbare Hardwarekomponenten: Selbstrekonfiguration eines Xilinx Virtex-4 FPGAs”. Bachelor Thesis (Studienarbeit). Germany: Carl von Ossietzky Universität Oldenburg, 2006.