CARL
VON
OSSIETZKY
universität OLDENBURG
FAKULTÄT II
INFORMATIK, WIRTSCHAFTS-
UND RECHTSWISSENSCHAFTEN

Verification Architectures for Complex Real-Time Systems

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inform. Johannes Faber

Oldenburg, 19. April 2011

Gutachter:	Prof. Dr. E.-R. Olderog Prof. Dr. B. Finkbeiner (Uni Saarbrücken)

Datum der Einreichung: 19.04.2011
Datum der Verteidigung: 29.08.2011

© 2011 by the author

Author's address:
Johannes Faber
Fakultät II, Department für Informatik
Abteilung „Entwicklung korrekter Systeme"
26111 Oldenburg
Germany

E-mail: johannes.faber@informatik.uni-oldenburg.de

For my loved girls
Melanie & Elyenna

Abstract

In the analysis of complex real-time systems, several aspects have to be covered, e.g., behaviour that conforms to communication protocols, rich data structures, and timing constraints such that the system reacts timely to external events. In practise it turns out that to handle such systems engineers fall back on combinations of techniques. An example is the Unified Modelling Language, combining multiple graphical notations for different views of a design. Similarly, in the world of formal analysis there has been a lot of work on integrating specification techniques to condense advantages of single formalisms into combined formalisms. However, a major problem remains: these integrated techniques are designed for heterogeneous systems, and when formally verifying those systems, one has to cope with their inherent complexity.

To solve this problem, we propose to verify global properties by combining local analyses using abstract, behavioural protocols. This idea originates from previous case studies, where a specification of the European Train Control System is decomposed according to a behavioural protocol. The protocol splits the system runs into several phases with local real-time properties which hold during these phases. After showing the correctness of a desired global safety property, this property is also guaranteed by all instances of the protocol that fit to the structure of the protocol and satisfy all local properties. We generalise and formalise this approach in a uniform framework for combined real-time formalisms.

As several combined specification formalisms are based on Communicating Sequential Processes (CSP), we introduce a CSP extension to specify system protocols with data and unknown processes to capture a large class of systems. On the unknown processes, local real-time assumptions are specified by formulae in an arbitrary logic. By this means, a decomposition of a global property into local assumptions on protocol phases is defined. We call these protocols with local real-time assumptions *Verification Architectures (VA)*. To establish safety properties on VAs, we embed the CSP extension into a temporal dynamic logic and introduce a sound sequent-style proof calculus. We prove that all models that structurally refine the CSP part of a VA and that satisfy the real-time assumptions inherit the desired properties.

The instantiation of VAs is exemplified with a combined formalism, called CSP-OZ-DC. The real-time logic Duration Calculus is used to specify assumptions on protocol phases. We introduce a syntactical proof rule to show efficiently that a CSP-OZ-DC specification structurally refines a VA protocol. The correctness of the assumptions is shown by using an existing model checking approach. Furthermore, by presenting a case study from the European Train Control System, we demonstrate the verification of a complex real-time system with the VA approach.

Zusammenfassung

Für die Analyse komplexer Realzeitsysteme sind verschiedene Faktoren relevant und müssen berücksichtigt werden. Dazu gehören unter anderem Kommunikations- und Kontrollflusseigenschaften, die verwendeten Datenstrukturen und das zeitliche Verhalten des Systems. Um diese heterogenen Aspekte umfassend beschreiben zu können, hat es sich in der Praxis durchgesetzt, Kombinationen von Analysetechniken zu verwenden. Ähnliches gilt für die formale Analyse und Verifikation von Realzeitsystemen, für die ebenfalls verschiedene formale Techniken kombiniert werden. Da diese Systeme, die mit kombinierten Techniken untersucht werden, inhärent komplex sind, gerät die formale Analyse an die Grenzen ihrer Möglichkeiten – es müssen Wege geschaffen werden, die Komplexität der Systeme zu reduzieren.

Aus diesem Grund wird in dieser Arbeit ein Ansatz vorgeschlagen, der die Analyse globaler Systemeigenschaften auf lokale Eigenschaften reduziert. Dies geschieht mit Hilfe von formalen Entwurfsmustern, die abstrakte Protokolle für bestimmte Systemklassen beschreiben. Die Protokolle unterteilen Systemabläufe in verschiedene Phasen, für die Realzeiteigenschaften als Annahmen vorausgesetzt werden. Wenn gezeigt werden kann, dass das Protokoll gewünschte Sicherheitseigenschaften gewährleistet, dann gelten diese Eigenschaften für alle Instanzen des Protokolls; dazu müssen Struktur von Instanz und Protokoll übereinstimmen und die lokalen Realzeiteigenschaften nachgewiesen werden. In dieser Arbeit wird dieser Ansatz mit Fokus auf kombinierte Spezifikationstechniken untersucht.

Da verschiedene kombinierte Formalismen auf der Prozessalgebra Communicating Sequential Processes (CSP) basieren, wird eine Erweiterung von CSP um Daten und unbekannte Prozesskomponenten vorgestellt, mit der abstrakte Systemprotokolle mit großem Freiheitsgrad beschrieben werden können. Für die unbekannten Prozesskomponenten können Realzeitannahmen in einer beliebigen Logik definiert werden. Da sich auf diese Weise globale Eigenschaften des Protokolls in lokale Annahmen zerlegen lassen, werden solche Protokolle als *Verifikationsarchitekturen (VA)* bezeichnet. Um globale Eigenschaften von VAs nachweisen zu können, werden eine Einbettung der CSP-Erweiterung in Dynamische Logik und ein Sequenzenkalkül für diese Logik präsentiert.

Die Instanziierung von VAs wird exemplarisch für die kombinierte Sprache CSP-OZ-DC gezeigt, indem eine Beweisregel für einen syntaktischen Verfeinerungsnachweis eingeführt wird. Die Korrektheit der lokalen Eigenschaften wird mit existierenden Model-Checking-Verfahren gezeigt.

Abschließend wird mit Hilfe einer Fallstudie über das Europäische Zugkontrollsystem belegt, dass der VA-Ansatz geeignet ist, komplexe Realzeitsysteme zu verifizieren.

Contents

1. Introduction **1**
1.1. Running Example: A Small Train Control System 5
1.2. Overview of the VA Approach 6
1.3. Related Work on Pattern-Based and Compositional Verification 8
1.4. Structure of this Thesis 11
1.5. Acknowledgements 12

2. Preliminaries **13**
2.1. CSP, OZ, and DC 14
2.1.1. Communicating Sequential Processes 14
2.1.2. Object-Z 17
2.1.3. Duration Calculus 22
2.2. Combining CSP, OZ, and DC Into a Parametric Specification Language 26
2.2.1. Syntax of CSP-OZ-DC 28
2.2.2. Semantics of CSP-OZ-DC 30
2.2.3. Related Combined Approaches 32
2.3. Phase Event Automata 33
2.3.1. Syntax and Semantics of Phase Event Automata 33
2.3.2. Operational CSP-OZ-DC Semantics in Terms of PEA 38

3. Extended CSP for Verification Architectures **41**
3.1. CSP Processes for Verification Architectures 42
3.1.1. CSP Processes with Data Constraints 43
3.1.2. Unknown Processes 47
3.1.3. Assumptions on Unknown Processes 49
3.1.4. Running Example: A Train Control Protocol 50
3.2. Properties of Extended CSP 52
3.2.1. Continuous CSP Operators 52
3.2.2. Discontinuity of Parallel Composition 54
3.3. Normal Forms 56
3.4. Discussion 60
3.4.1. Parametric Systems 60
3.4.2. Semantics 61
3.4.3. Related work 62

4. A Sequent Calculus for Verification Architectures **65**
4.1. Dynamic Logic over CSP Processes with Data 66
4.2. Sequent Calculus 68
4.3. Proof Rules 70
4.3.1. Structural Rules 70
4.3.2. Propositional Rules 70
4.3.3. First-Order Rules 71
4.3.4. Symbolic Execution of dCSP Formulae 72
4.3.5. Symbolic Execution of dCSP Specifications with Unknown Processes 75
4.3.6. Induction Rules 77
4.3.7. Auxiliary dCSP Rules 79
4.4. Soundness of the Calculus 80
4.5. Embedding of a Real-Time Logic 93
4.5.1. Checking the Side-Conditions for the Box Operator 93
4.5.2. Checking the Side-Conditions for the Diamond Operator 95
4.6. Discussion 99
4.6.1. Discussion of the VA Approach 99
4.6.2. Related Work 100

5. Refinement of Verification Architectures **105**
5.1. Refinement of Verification Architectures 105
5.2. Simulation of Processes 109
5.3. Proof Rules for Checking Refinement 110
5.4. Syntactical Proof Rule for Process Refinement 114
5.5. Property Inheritance 119
5.6. Discussion 122

6. Limitations and Extensions **127**
6.1. Parallel Unknown Processes 128
6.1.1. An Automata-Based Semantics for Parallel Unknowns 129
6.1.2. Translation-Based Approaches 132
6.1.3. Rely-Guarantee Reasoning for Parallel Unknowns 140
6.1.4. Instantiating Parallel Unknowns 146
6.1.5. An Interpretation-Based Semantics for Parallel Unknowns . . . 149
6.2. Verifying Timing Properties 158
6.2.1. Test Processes for Timing Properties 159
6.2.2. Extended Calculus for Timing Properties 161
6.3. Examining Completeness 163
6.3.1. Completeness of VA Language 164
6.3.2. Completeness of Local Assumptions 166
6.4. Complementary Decomposition Techniques 169
6.4.1. Slicing Formal Specifications 170

6.4.2. Layered Composition for Timed Protocols 171

7. Implementation and Tools 173
7.1. Syspect 174
7.1.1. UML Profile for Real-Time Systems 174
7.1.2. Tool Structure 178
7.1.3. Syspect Plug-Ins 179
7.2. Verification with Syspect 181
7.2.1. Transition Constraint Systems 181
7.2.2. Verification of Syspect Specifications 182
7.2.3. Slicing CSP-OZ-DC Specifications in Syspect 185
7.2.4. Further Verification Plug-Ins 185
7.3. Syspect Verification Architecture Plug-In 185
7.3.1. Modelling of VAs 185
7.3.2. CSP-OZ-DC Representation of a VA 186
7.3.3. Verification of VAs 187
7.4. Discussion 189

8. Case Studies 191
8.1. Running Example: Small Train Control System 191
8.1.1. Verification of the Architecture 192
8.1.2. Instantiation by a CSP-OZ-DC Model 194
8.2. European Train Control System 196
8.2.1. Case Study Scenario: Emergencies in Train Control Systems . 197
8.2.2. Previous ETCS Case Studies 198
8.2.3. VA for the ETCS Case Study 202
8.2.4. VA Verification 207
8.2.5. Instantiating the VA 214
8.2.6. Checking the Instantiation with Syspect 216
8.2.7. Discussion 220

9. Conclusions 223
9.1. Discussion 224
9.2. Alternative Approaches 227
9.3. Perspectives 228

A. Case Study Material 233
A.1. Train Control System of the Running Example 233
A.1.1. Proof Tree for the VA 233
A.1.2. CSP-OZ-DC Model of the Train Control System 240
A.1.3. CSP-OZ-DC Representation of the VA 243
A.1.4. Alternative Architecture without eCSP 244
A.2. ETCS Emergency Message Case Study 247

A.2.1. Original CSP-OZ-DC Specification from [MFHR08] 247
A.2.2. VA Proof Tree for the ETCS Case Study 250
A.2.3. Modified CSP-OZ-DC Specification Matching VA 257

Bibliography **263**

Sequent Rules **279**

Glossary of Symbols **281**

Index **287**

List of Figures

1.1. Context of this thesis 3
1.2. Scenario of running example 5
1.3. Illustration of a Verification Architecture 7

2.1. Exemplary CSP-OZ-DC specification 27
2.2. Exemplary Phase Event Automaton 36
2.3. A PEA representing a process of Fig. 2.1 39
2.4. Translation of the OZ part into PEA 40

4.1. Proof tree for Example 4.3.2 78
4.2. Protocol structures to be handled with the DL approach 99

6.1. Translation of unknown processes into PEA 132
6.2. Sequent-style proof rules for timing properties 160
6.3. Generic CSP-OZ-DC specification *cod* 165

7.1. An exemplary class diagram 175
7.2. State machine for Train 176
7.3. Exemplary component diagram 176
7.4. Screenshot of Syspect 178
7.5. Syspect verification toolchain 180
7.6. Counterexample view in Syspect 183
7.7. VA of the running example modelled in Syspect 186
7.8. Tool chain for VA plug-in 187

8.1. Scenario of the small train control example 192
8.2. VA for the train control system 193
8.3. Syspect class diagram for the train control system 194
8.4. Part of the train specification 195
8.5. Case study scenario 197
8.6. Components of the Case Study 197
8.7. Signature for the ETCS VA 203
8.8. The VA process ETCS-EM for the ETCS case study 204
8.9. The VA process as modelled in Syspect 205
8.10. DC assumptions on the protocol phases 206
8.11. Representation of the parallel composition without unknown parts . . 210

8.12. Representation of *ProcFree* without constraints over clocks 210
8.13. Proof tree for timing constraints 212
8.14. Side-conditions to be checked for (T1) up to (T3) of page 212 213
8.15. Syspect class diagram for the ETCS case study 214
8.16. Control structure of class *Trck* as modelled in Syspect 215
8.17. Class diagram with *Trck* as protocol class 216
8.18. Process representing the control structure of the ETCS model 217

1 Introduction

> People want to forget the impossible. It makes their world safer.
>
> *(Silas, in* The Graveyard book, *Neil Gaiman)*

1.1. Running Example: A Small Train Control System 5
1.2. Overview of the VA Approach 6
1.3. Related Work on Pattern-Based and Compositional Verification 8
1.4. Structure of this Thesis 11
1.5. Acknowledgements 12

Systematic analysis of software and hardware is an important instrument to increase the safety of computer systems, and this is particularly relevant when designing systems in safety critical application domains like transportation systems or industrial plants whose malfunction may endanger monetary investments, environment, or life. For this reason, there is a lot of ongoing research with the aim of helping system engineers to develop high-quality software or hardware that is guaranteed to answer the desired purpose. There are two major research directions to tackle these issues.

On the one hand, *informal software engineering approaches* aim at methods and tools to capture the complexity of a system which usually can only be handled based on the division of labour. Systematic and standardised techniques are investigated, like development processes, architectural concepts, modelling languages, and quality control. The main focus of those techniques is to increase the quality of complex software systems during the entire development and life cycle.

On the other hand, *formal analysis techniques* are also targeted to increase the quality of software systems, but these techniques approach from the side of rigor-

ously mathematical methods. Systems are described by languages that are often not focused to convenient applicability when developing large software systems, rather having a well-defined semantics in a mathematical domain that can be exactly analysed. Ideally, those analyses are performed completely automatic with the help of sophisticated verification tools. To cope with the increasing complexity of software systems, abstraction techniques and decompositional methods are applied for reducing the state space, particularly for infinite state systems, and for partitioning verification problems into smaller parts.

In this thesis, we integrate techniques of both worlds that have been proven to be successful in their application domain, in order to bundle the advantages of the approaches and to overcome specific difficulties from which both suffer.

Informal analysis of complex real-time systems. The subject of this work are basically complex, safety-critical systems, i.e., systems for which specific safety properties are indispensable. This includes plenty of daily-life software systems: for instance, automated cruise-control systems or safety systems of modern vehicles, medical equipment, power plants; wherever technical systems take over control from humans. Suchlike systems are often inherently complex, because they are determined by orthogonal system dimensions. When analysing this complexity, it turns out that there is hardly a single formalism that adequately describes all relevant aspects. Hence, in practice system engineers often fall back on combinations of different techniques for system analysis. To cope with heterogenous system dimensions, the *Unified Modelling Language* (UML) [RJB99, Dou97, Dou04] has been developed, combining multiple graphical notations for different views of a design. But even though UML meets the identified requirement to comprehensively capture the system dimensions of complex systems by providing diagrammatic modelling techniques and even though it is well-integrated into industrial software development processes, it lacks an important feature that is necessary for the integration into *formal* software analysis: an exact semantics. In practice, this leads to misconceptions and misunderstandings in software development, and UML cannot directly be used in combination with formal verification techniques.

A further concept to reduce the complexity of software designs is the use of *design patterns* [GHJV95, SSRB00, Dou02], which allow developers to structure large design problems into smaller parts for which standardised solutions exist. Once a solution to a dedicated design problem is realised with patterns, they can be reused in different contexts. However, similarly to UML, pattern-based approaches are usually applied informally, often described by textual, tabular, or graphical representations. Thus, suchlike pattern-based techniques are generally incompatible with formal analysis.

Formal analysis of complex real-time systems. In the world of formal analysis and verification there has been a lot of work on integrating specification techniques to condense advantages of single formalisms into combined formalisms, e.g., [But92,

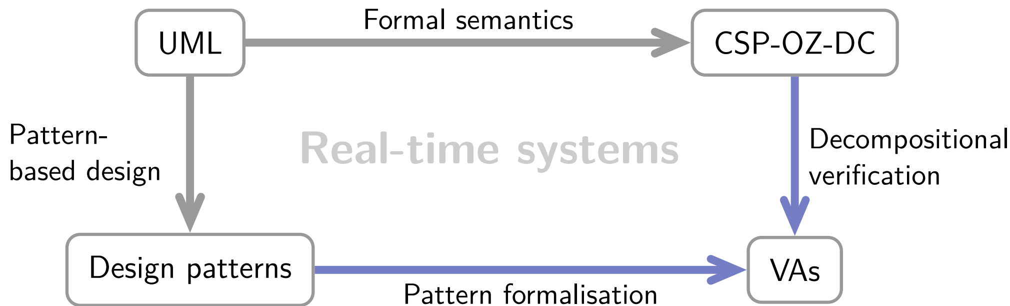


Figure 1.1.: Context of this thesis

RAI92, MD98, AM98, Fis00, WC01, Süh02, SLD08]. In the analysis of safety-critical real-time systems, several important aspects have to be covered by a formalism in order to cope with the complexity of the system:

1. behaviour that conforms to communication protocols and the internal control flow of components,
2. rich data structures with possibly infinite domains, e.g., linked lists or real numbers, and
3. timing constraints such that the system reacts timely to external events.

An example for such a combined specification formalism is *CSP-OZ-DC*, which was developed by Hoenicke and Olderog [HO02a, HO02b, Hoe06]. CSP-OZ-DC integrates the well-investigated formalisms *Communicating Sequential Processes (CSP)*, to specify the control flow, *Object-Z (OZ)*, to define the data space and data changes, and *Duration Calculus (DC)*, to impose dense real-time constraints on system specifications. In [Hoe06, MFHR08] an operational semantics and a model checking approach for CSP-OZ-DC are presented, such that CSP-OZ-DC can be used for formal analysis of real-time systems with automated methods. Furthermore, in [MORW08, FLOQ11] CSP-OZ-DC is integrated into the software engineering process with UML by providing a UML profile for real-time systems. This profile defines a subset of UML diagrams, in which the elements hold certain additional annotations; a translation of this profile into a CSP-OZ-DC semantics is given. By this means, real-time systems can be specified with UML and formally analysed and verified with CSP-OZ-DC.

However, a major problem remains: suchlike integrated formal techniques, for which CSP-OZ-DC is an example, are designed for heterogeneous and thus inherently complex systems, and when formally verifying those systems, we have to cope with this complexity. Even though integrated specifications can often be translated into leaner formalisms, these still need to cover all relevant information. Particularly, systems with a large degree of concurrency often suffer from the state space explosion problem. Thus, formal analysis of those systems is only applicable if they are decomposed in a suited manner into smaller parts.

Idea of the Verification Architecture approach. To solve the identified complexity issues for heterogenous real-time systems, we carry the ideas of informal design patterns over to formal specifications. To this end, a formalisation of patterns for complex real-time systems, so-called *Verification Architectures (VA)*, is introduced. Moreover, we examine how these VAs are verified with respect to desired safety properties, and how they enable compositional verification of complex systems specified with combined specification languages. Figure 1.1 gives an overview of the context of this work and the interrelation of informal models (represented by 'UML' in the diagram) and design patterns in contrast to formal specifications (represented by 'CSP-OZ-DC') and Verification Architectures.

The benefits of the Verification Architecture approach are twofold: First, analogous to design patterns, VAs can be reused in different applications. In doing so, verified properties of the VA are inherited by all instances. Second, VAs give rise to a decomposition of the system: global properties are verified by combining local analyses. This idea originates from previous case studies [MFHR08] in the context of the transregional research centre AVACS[1], where a specification of the European Train Control System (ETCS) [ERT02] was decomposed according to its abstract behavioural protocol. The protocol splits the system runs into several phases (e.g., braking phase and running phase) with local real-time properties that hold during these phases. After showing the correctness of a desired global property for the protocol, this property is also guaranteed by all instances of the protocol that satisfy the local properties. Since the local properties correspond only to parts of the protocol, it is more efficient to verify the local properties instead of directly checking the global safety property. In case of the ETCS case study, the latter was not possible due to the size of the case study model, while verification of the local properties was successful [MFHR08]. We generalise and formalise this approach in the context of combined languages. The approach is structured into several layers:

1. abstract behavioural *protocols with unknown parts*, which have a large degree of freedom to comprise a large class of concrete systems, need to be specified and verified with respect to desired *global safety properties*;

2. since the analysed systems are often time-dependent, it is important to allow imposing of additional *local real-time assumptions* on protocol phases, and it must be possible to verify the protocols taking these assumptions into account;

3. it needs to be checked that concrete systems, given as combined specifications to capture heterogeneous systems, are *instances* of the protocol;

4. it needs to be checked that concrete instances actually *guarantee the local real-time assumptions* on the protocol phases.

The challenge is to tackle each layer of the problem with a suitable formalisation and to integrate the heterogeneous formalisations into a uniform framework.

[1] `http://www.avacs.org`

Summary of contributions. We introduce a pattern-based Verification Architecture approach for complex (dense) real-time systems, a new conceptional approach on how to use behavioural protocols as a decomposition technique to enable verification of systems specified by combined formalisms. By establishing behavioural design patterns for suchlike formalisms, we extend existing work on integrating combined formalisms and software engineering processes. In particular, when using VAs in combination with CSP-OZ-DC, we benefit from its descriptive UML representation and graphical tool support.

Beyond this conceptional contribution, we present a theoretical model based on CSP for the specification of VAs. The verification approach for VAs makes use of theorem proving as well as model checking: an embedding into Dynamic Logic and a sequent-style proof calculus for the VA model are introduced, whereas local assumptions are verified with existing model checking techniques. A refinement notion for VAs is established that is used to automatically prove that a concrete model instantiates a VA, by which the model inherits all safety properties of the VA. Finally, we demonstrate our approach using a case study from the AVACS sub-project R1.

In the following, the basic technical details of the VA approach are introduced. But beforehand, we present a running example that is used throughout this thesis.

1.1. Running Example: A Small Train Control System

The emerging European Train Control System (ETCS) is an international standard [ECS99, ERT02] that shall replace national train control systems to ensure cross-border interoperability and to improve railway safety as well as track utilisation. In the final ETCS implementation level, the existing national trackside systems for detection of train speed, location, and integrity will not be used anymore. Instead, a radio block centre (RBC) controls the traffic in a well-defined area. It ascertains speed and position values in cooperation with the ETCS on-board units of the trains. RBCs and trains communicate over a GSM-R radio connection. One objective of the ETCS is to increase the possible traffic density. For this purpose, the *moving block principle* is used, by which a *movement authority* is always granted up to a position closely behind the preceding train. In case of an accident, the train control system has to stop all trains safely.

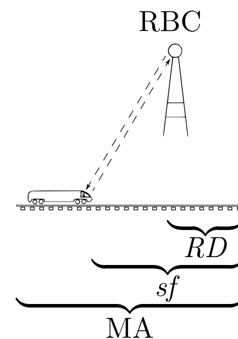


Figure 1.2.: Example

The setting is pictured in Fig. 1.2: an RBC grants movement authorities (MAs) to a train. The system is considered safe as long as the train stays within the MA. The distance of the train to the end of the MA is given by a real-valued variable sf, reflecting the safety of the system, that shall never be below 0. The *reaching distance* position RD is the last position at which the train needs to apply the brakes to stop

in time. To ensure safety, the system shall comply with a simple protocol structuring all runs into phases: The first protocol phase *FAR* describes the situation where the train is in a safe mode with a large distance to the end of the MA. In a second phase *CHK*, the train is required to check the distance to the end of the authority and to take an appropriate action: either it may request an extension of the MA (*REQ*) or—if the train is too close to the end of the MA—it changes to a recovery phase *REC* in which a counteraction is initiated, e.g., application of emergency brakes in order to stop the train safely.

This is a typical protocol that can be realised as a VA: it comprises some defined behaviour (for instance, the decision which is necessary when the train is too close to the end of the MA), real-time constraints (the train has to check the situation within certain time intervals), and some unknown behaviour that is not relevant for the protocol (it does not matter how the train behaves in the *FAR* phase as long as the train is not too close to the end of the MA).

This example is to be continued during this thesis and we will show how such protocols are specified, verified, and instantiated.

1.2. Overview of the VA Approach

We now introduce the ideas and basic formalisations of our approach to use patterns for the compositional verification of real-time systems.

Generic protocol. A Verification Architecture (VA) is a formalisation of a generic protocol with local real-time assumptions that ensures the safety of a class of systems. We use parametric *CSP processes with data constraints and unknown parts* to specify VAs. A VA consists of a parametric CSP process, enriched by data constraints, and additionally, of assumptions on specific phases of the generic protocol, defined as formulae in an arbitrary real-time logic. In our examples, we will use Duration Calculus (DC) to specify local assumptions. Figure 1.3 depicts a VA in the dashed box. The locations represent phases of a protocol (for instance, the protocol phases *FAR*, *CHK*, *REQ* and *REC* of the running example) with additional local assumptions asm_1 up to asm_4.

The name *Verification Architecture* reflects the purpose of the protocol, which is dedicated to formal verification by decomposing a global property into local properties of protocol phases. We use the terms *Verification Architecture* and *protocol* (that is formalised as VA) synonymously. If we refer only to the structure of the protocol process without the local assumptions, we use the notion *protocol structure*.

Proof rules. We use a rule-based sequent calculus to establish the validity of global properties (e.g. *collision freedom* for the running example in Sect. 1.1) for VAs. To this end, we have to take into account the timed assumption on the protocol phases as well as the data constraints.

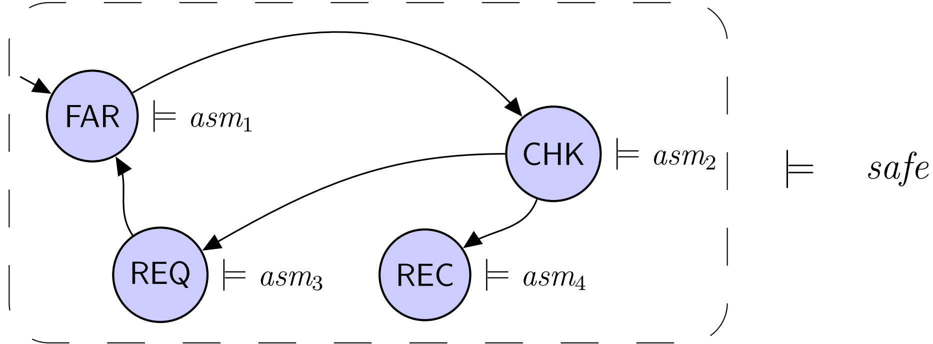


Figure 1.3.: Illustration of a Verification Architecture

Concrete instantiations. We consider concrete instances of systems that conform to VAs. We show when such instances actually refine the process structure of the corresponding VA. To this end, we present a proof rule that allows us to check syntactically that the structure of a concrete CSP-OZ-DC specification conforms to a VA.

Correctness of local assumptions. One has to prove that every process of a concrete instance that refines a protocol phase of the VA satisfies every corresponding local assumption of the phase. For instance, a process instantiating the *FAR* phase of the example is required to satisfy asm_1, which is a property stating that the train is not moving too fast. This step is done with a suited verification technique for the logic of the assumptions. For example, in the case of DC and CSP-OZ-DC the verification approach of [MFHR08] is used. The local assumptions only depend on parts of the CSP-OZ-DC model and, thus, the resulting verification tasks are simpler than the global safety property.

Guaranteed correctness. Finally, if the structural refinement relation and the local assumptions are valid, our VA approach guarantees that the global safety property is also valid for the entire model.

Formalisation of the VA Approach

We give a formalisation of the problem. Let $prtcl(\overline{p}, P_1, \dots P_n)$ be an abstract behavioural protocol depending on a vector of data parameters $\overline{p}$ and on process parameters P_i. Additionally, we consider temporal assumptions $asm_1(\overline{p}), \dots, asm_n(\overline{p})$ on the P_i, which may also depend on the data parameters $\overline{p}$. We denote the combination of protocol structure and temporal assumptions as *Verification Architecture* (VA). Our aim is to show that a safety property $safe(\overline{p})$ is valid for every possible model which is a refinement of the behavioural protocol and which respects the assumptions.

To apply our approach, we have to show that the VA is correct, i.e., the protocol is correct for all parameters and processes respecting the assumptions:

$$\forall\, \overline{p}, P_i \bullet \left(\bigwedge_{i=1,\dots,n} P_i \models asm_i(\overline{p}) \right) \Rightarrow (prtcl(\overline{p}, P_1, \dots, P_n) \models safe(\overline{p})) \tag{1.1}$$

We will provide proof rules for the verification of this proof task. Once it is verified, this result is reusable as all instances of this architecture inherit the correctness property automatically. We only have to show that it is a refinement of the protocol structure and that the local assumptions are valid, which is due to their locality easier than to verify the global property directly. We consider an instance $spec(\overline{p}, P_1^0, \dots, P_n^0)$ of the abstract protocol, where the P_i^0 are instances of the process parameters. First, we have to show that every trace of the instance is also a trace of the protocol:

$$\forall\, \overline{p} \bullet [\![spec(\overline{p}, P_1^0, \dots, P_n^0)]\!] \subseteq [\![prtcl(\overline{p}, P_1^0, \dots, P_n^0)]\!]. \tag{1.2}$$

This relation is shown syntactically for a specific class of instances (cf. Sect. 5.1). Second, we have to show that the assumptions are valid for the concrete specification:

$$\forall\, \overline{p} \bullet P_i^0 \models asm_i(\overline{p}) \text{ for all } i \in 1..n. \tag{1.3}$$

This is done by applying existing verification techniques for the language of the assumptions. With this, our approach yields that the desired safety property is valid for the concrete model. We argue that this proposition is correct. From (1.1) and (1.3) we can conclude (1.4), and with (1.2) we get the desired property (1.5).

$$\forall\, \overline{p} \bullet prtcl(\overline{p}, P_1^0, \dots, P_n^0) \models safe(\overline{p}) \tag{1.4}$$

$$\forall\, \overline{p} \bullet spec(\overline{p}, P_1^0, \dots, P_n^0) \models safe(\overline{p}) \tag{1.5}$$

We summarise that if a correct VA is given, we only have to show that (1) a concrete model is actually a structural refinement of the abstract protocol, and (2) the model respects the local assumptions. Then we can conclude the correctness of the entire model.

1.3. Related Work on Pattern-Based and Compositional Verification

We present a formal and generic framework for reusable verification patterns that are used to decompose a specification and to simplify verification of complex systems. To this end, the VA approach involves formal analysis on several stages:

1. behavioural protocols (comprising data aspects and allowing for a large degree of freedom) with a formal semantics

2. verification of these protocols (proof calculus)
3. refinement by concrete specifications
4. combination with higher-level specifications (e.g., CSP-OZ-DC and the UML profile for CSP-OZ-DC)
5. formal verification of local assumptions (e.g., by model checking)
6. inheritance of safety properties

Other works on formal design patterns often focus on single aspects or on handling of standard design patterns (considering static analysis of code and structures in object-oriented languages) in contrast to the behavioural, protocol-based pattern we consider here. In the following, we review related pattern-based and decompositional approaches.

Design Patterns. Our approach is inspired by [DHO06], where for a fixed DC protocol, a design pattern for cooperating traffic agents is introduced. The work [KRS07] presents Design Verification Patterns, which are also motivated by [DHO06]. These patterns are introduced as a formal counterpart to the well-known design patterns from classical software engineering. [KRS07] proposes to use Rely-Guarantee pairs as proof obligations to characterise patterns and exemplary gives such pairs for two design patterns. But contrary to the Verification Architectures approach, no formal framework for the use and application of design patterns is introduced. It is neither examined how to verify the patterns itself nor exploited that the pattern may lead to a natural decomposition for verification. Moreover, no tool support for automated verification is provided.

A general overview on formalisation techniques for design patterns is contained in [Tai07], but without considering verification of real-time systems. Instead, several formalisation approaches and languages are listed, basically with the aim to avoid the ambiguity of standard design patterns. An example is [FCA07], where object-oriented design patterns, like the Observer pattern [GHJV95], are specified in terms of the RAISE Specification Language [RAI92]. Other works with a similar focus to design patterns for static organisation of specifications, can be found in [SH04, BH07]. On the contrary, we consider behavioural protocol patterns used for formal verification of real-time systems. The work [HKM07] is more related to our approach: they also consider behavioural design patterns, which are specified in the experimental Ocsid language and verified by theorem proving. The desired properties are specified with linear-time temporal logic. The patterns are used for synthesis of instances, whereas we use a syntactic refinement check to establish a connection between pattern and instance. Additionally, they do not examine real-time specifications and do not present a formal framework for the decompositional verification of complex systems using design patterns.

Pattern-based approaches for real-time systems, e.g., [TZY+03, DHQ+04, KC05, FVVC06], often introduce timing patterns for bounded response, task periods, etc., contrary to our work, where arbitrary real-time properties are imposed on protocols.

A behavioural design pattern method that is similar to ours is realised in the context of the Fujaba tool suite [GTBF03, Gie03]: design patterns are specified with discrete-time state charts without data and syntactically instantiated by refinement. This work is reviewed in more detail in Sect. 7.4.

Compositional methods. To establish global properties of a VA, their correctness has to be shown under certain local assumptions over unknown components. When verifying that a concrete specification is an instance of the VA, these assumptions have to be checked such that the instance is guaranteed to inherit the desired global property. By this means, the VA approach is similar to classical Assume-Guarantee (A-G) reasoning (in the context of combined, parametric specifications), where properties over components are shown under the assumption that the environment satisfies specific 'Assume' properties. A-G reasoning was initially introduced in the early 80s in [MC81, Jon81] and has been developed further in several works, e.g., [AL93, AL95, FMS98, MCF+97]. The work [dRdBH+01] contains a general introduction into A-G reasoning without time and without the context of conjoint verification techniques.

In contrast to our approach, classical A-G approaches usually consider components together with assumptions on the environment of a component, i.e., components are analysed with respect to a parallel environment. In the VA approach, assumptions over unknown protocol parts are integrated into more complex protocol structures, also incorporating sequential and parallel composition as well as recursion. However, existing A-G methods can be used to complement our approach to solve properties over parallel unknown components. In particular, we apply *Rely-Guarantee* reasoning in Sect. 6.1.3, which is a sub-form of A-G reasoning dedicated to shared variable systems; we adapt Rely-Guarantee reasoning for the handling of properties of parallel unknown components. In a similar way, it will be possible to adapt other existing A-G methods to the VA approach.

In [MWW08, Met10] an A-G verification approach for CSP-OZ (without time) is presented that does not consider decompositions by given protocols but instead uses a learning-based algorithm to generate assumptions on layered components.

An approach with basic ideas similar to ours was developed by D'Errico and Loreti [DL09, DL10]: they investigate concrete CCS processes [Mil80] with an unknown environment that is constrained by Hennessy-Milner logic [HM85]. They provide a compositional proof system based on A-G reasoning for these processes and introduce a notion of property-preserving refinements of the unknown parts. The main difference to the work presented here is that they are focused on simple, only event-based, CCS processes and Hennessy-Milner logic and consider no more complex or combined specification languages and neither real-time properties nor data constraints.

1.4. Structure of this Thesis

In this first chapter, we have motivated the usage of the conceptual Verification Architecture approach in the context of informal and formal models of software systems. We have introduced the basic idea of VAs to use abstract behavioural protocols to structure verification tasks. By this means, VAs are used as a decomposition technique to enable verification of complex systems specified by combined formalisms.

- In Chap. 2, we preliminary present basic formalisms that are needed throughout this work. These are the combined formalism CSP-OZ-DC and Phase Event Automata, a timed automata model for the operational semantics of CSP-OZ-DC.

- Chapter 3 introduces a new CSP dialect *eCSP* that extends classical CSP by data constraints and unknown processes with local real-time assumptions for the specification of VAs. Furthermore, we analyse semantical properties of this CSP dialect and present two useful normal forms.

- To prove properties over eCSP, it is embedded into Dynamic Logic in Chap. 4. A sequent calculus for this logic is introduced and proven correct.

- In Chap. 5, we establish a notion of refinement for eCSP and for combined specifications in terms of CSP-OZ-DC. Moreover, we present efficient, syntactical rules to prove that a CSP-OZ-DC specification is a structural refinement of an eCSP process, and we analyse property inheritance for the refinement relation.

- Limitations and possible extensions of the VA approach are discussed in Chap. 6. In particular, we examine how parallelism over unknown parts is solved and present a second semantics for eCSP that is suited to adequately describe parallelism over unknowns. We compare the different semantics of eCSP and discuss shared variable access in eCSP. In addition, we shortly analyse verification of timed properties and completeness issues. An examination of decomposition techniques that complement the VA approach concludes the chapter.

- Chapter 7 shows how the VA approach is embedded into a software engineering process: a graphical UML tool is presented that supports Verification Architectures and the UML profile for CSP-OZ-DC.

- The application of the VA approach is demonstrated in Chap. 8 by two case studies: the train control system of the running example and a larger ETCS case study from the AVACS sub-project R1.

- Finally, Chap. 9 concludes this thesis with a general discussion and an evaluation of the VA approach.

- In the appendix, the reader finds a glossary of symbols and an index, which might be helpful while reading this thesis. Furthermore, a list of all rules of the introduced sequence calculus and auxiliary case study material are provided.

Sources

Some parts of this thesis are based on previously published works by the author (most of them significantly revised):

- Section 1.2 is partly taken from [Fab10a].
- Parts of Chap. 2 contain adapted material from [FS07] (Sect. 2.1.1), [FJSS07, MFHR08] (Sect. 2.3), and from [FLOQ11] (the part about the CSP-OZ-DC translation into PEA).
- Chapter. 3 is based on [Fab10b, Fab10a].
- Parts of Chap. 7 (Sect. 7.1 and 7.2) are taken from [FLOQ11], which is joint work with Sven Linker, Ernst-Rüdiger Olderog, and Jan-David Quesel.
- The running example (Sect. 8.1, App. A.1) was presented in [Fab09, Fab10b].
- The ETCS case study in Sect. 8.2 and the CSP-OZ-DC model in App. A.2.1 are based on case study material in [MFR06, FM06, MFHR08], which is joint work with Roland Meyer, Jochen Hoenicke, and Andrey Rybalchenko, but the VA and the modified ETCS model are presented here for the first time.

1.5. Acknowledgements

This thesis would not have come to fruition without many people who contributed in one or another way to this work: current and former colleagues of the working group *Correct System Design*, who provided me with a very friendly working atmosphere, useful hints and encouragement; the students who implemented important parts of Syspect, the PEA toolkit, and Moby/PEA; colleagues from AVACS and Paderborn with whom I discussed my research and who developed the verification approaches used in this work and the tools ARMC and SLAB; and my friends and my family, who were listening to me for several years whenever I got stuck with my work.

I am deeply grateful to all of you!

I want to express my particular gratitude to Ernst-Rüdiger Olderog for giving me the opportunity and the necessary freedom to develop my research ideas and for always having an open office for me. I thank Bernd Finkbeiner for agreeing to act as a co-referee even in face of the coincidence with the AVACS review process.

Furthermore, I sincerely thank Melanie Faber, Henrik Lipskoch, Christian Schultz-Brummer, Esther and Caleb McDonald, and Jörn Störk for proofreading parts of this thesis.

2 Preliminaries

> You know something? We came here for you, a long time ago, when you died. Well, it wasn't here and that wasn't you, but we did anyway.
>
> *(Delirium, in* The Wake*, Neil Gaiman)*

2.1. CSP, OZ, and DC . . . 14
- 2.1.1. Communicating Sequential Processes . . . 14
- 2.1.2. Object-Z . . . 17
- 2.1.3. Duration Calculus . . . 22

2.2. Combining CSP, OZ, and DC Into a Parametric Specification Language . . . 26
- 2.2.1. Syntax of CSP-OZ-DC . . . 28
- 2.2.2. Semantics of CSP-OZ-DC . . . 30
- 2.2.3. Related Combined Approaches . . . 32

2.3. Phase Event Automata . . . 33
- 2.3.1. Syntax and Semantics of Phase Event Automata . . . 33
- 2.3.2. Operational CSP-OZ-DC Semantics in Terms of PEA . . . 38

The focus of this thesis is on the analysis and verification of complex real-time systems that are determined by heterogeneous system dimensions. To cope with such different dimensions, Hoenicke and Olderog introduced CSP-OZ-DC [HO02a, HO02b] that combines three well-investigated formalisms into a single language: it uses CSP [Hoa85, Ros98] to model the control flow of a system, Object-Z (OZ) [Smi92, Smi00] to specify data space and state changes via OZ schemata, and it applies a restricted

class of DC formulae [ZHR91, ZH04] for defining (dense) real-time constraints. CSP-OZ-DC is a declarative and object-oriented language. A key feature of CSP-OZ-DC is its separation of concerns, because every part, i.e., the control flow, the data space, and the timing part can be specified on its own. Its semantic is given in terms of interpretations, and it is compositional. Thus, if one can establish a safety property for a single part of the specification the property automatically holds likewise for the entire specification.

Due to this features, we make use of CSP-OZ-DC in the context of Verification Architectures, particularly, as specification language for concrete instances of architectures and for our case studies in Chap. 8. In this chapter, we introduce CSP-OZ-DC and a model for its operational semantics in detail.

2.1. CSP, OZ, and DC

2.1.1. Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a notation to describe concurrent and sequential processes. It was initially developed by C.A.R. Hoare [Hoa78, Hoa85]. Its basic building units are instantaneous events that can be arranged into processes using process algebraic operators, e.g., for sequential or parallel composition. We denote the set of all events by *Events* and the set of events occurring in a process P, the *alphabet of* P, by $alph(P)$.

Syntax of CSP

The CSP syntax is defined by the following BNF grammar:

$$\begin{aligned} P ::= {} & \texttt{Stop} \mid \texttt{Skip} \mid a \to P \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid \\ & P_1 \parallel_A P_2 \mid P_1 ||| P_2 \mid P_1 \mathbin{;} P_2 \mid P[R] \mid P \setminus A \mid X. \end{aligned}$$

We denote the so-defined set of process expressions with *Process*, $\mathit{Process} == P$. The process $\texttt{Stop}$ denotes a deadlocking process, $\texttt{Skip}$ a terminating process. For an event $a \in \mathit{Events}$, $a \to P$ denotes a process that communicates the event a and then behaves as the process P. The operators $\Box$ and $\sqcap$ represent external choice and internal choice, respectively. An external choice is resolved by the environment of the process, while internal choice is non-deterministically resolved by the process itself. Concurrent processes are described by $\parallel_A$ and $|||$. The former operator is used for processes that synchronise on events from a set $A \subseteq \mathit{Events}$, and the latter for interleaved processes without any synchronisation. The operator $\mathbin{;}$ denotes sequential composition of processes, i.e., the process behaves like P_1 until the process terminates and then it behaves like P_2. Hiding of events from a process can be specified with $\setminus$. The operator $P[R]$ renames the events in P according to a relation R on events. Finally, X stands for a process identifier, which must be declared with an expression

$X \stackrel{c}{=} P$ and which can be used to recursively define processes. Another representation of recursive processes is the fixed point expression $\mu\, X \bullet F(X)$.

Example 2.1.1. As example we consider the following CSP process equation system specifying a train component for the running example of Sect. 1.1.

$$
\begin{array}{rcl}
\mathtt{main} & \stackrel{c}{=} & (\mathit{extend} \rightarrow \mathtt{main}) \sqcap \mathit{FAR} \\
\mathit{FAR} & \stackrel{c}{=} & ((\mathit{InitialState}0 \parallel\!| \mathit{InitialState}1) \fatsemi (\mathit{check} \rightarrow \mathit{Checked})) \\
\mathit{InitialState}0 & \stackrel{c}{=} & \mathit{updSpd} \rightarrow \mathit{updPos} \rightarrow (\mathit{InitialState}0 \sqcap \mathtt{Skip}) \\
\mathit{InitialState}1 & \stackrel{c}{=} & (\mathit{sendCurPos} \rightarrow \mathit{InitialState}1) \sqcap \mathtt{Skip} \\
\mathit{Checked} & \stackrel{c}{=} & (\mathit{fail} \rightarrow \mathit{REC}) \,\square\, (\mathit{pass} \rightarrow \mathtt{main}) \\
\mathit{REC} & \stackrel{c}{=} & ((\mathit{applyEB} \rightarrow \mathit{RecCycle}) \fatsemi \mathtt{Stop}) \\
\mathit{RecCycle} & \stackrel{c}{=} & (\mathit{updSpd} \rightarrow \mathit{updPos} \rightarrow \mathit{RecCycle})
\end{array}
$$

This process equation system defines a process `main` as an internal choice between a process $\mathit{extend} \rightarrow \mathtt{main}$ and *FAR*. The latter is a process reference defined by the second equation of the system. The former is a prefix process, i.e., the process fires an event *extend* and behaves then as the `main` process again. The *FAR* process contains a sequential composition of two sub-processes, where the first is an interleaving of *InitialState*0 and *InitialState*1. The process *Checked* is defined as external choice: the process either receives a *fail* event from its environment and then behaves as process *REC* or it receives a *pass* and the `main` process is started again.

Semantics of CSP

There are three types of semantics that are commonly used to define the meaning of CSP processes: *operational semantics*, defining the semantics in terms of transition systems; *denotational semantics*, defining the semantics of a process in terms of a mapping into a semantical domain such that the semantics of a compound process can be computed by the semantics of its parts; and *algebraic semantics*, defining by algebraic laws which processes are meant to be equivalent. Roscoe [Ros98] gives an overview on these different semantical approaches to CSP.

In the class of denotational CSP semantics common representatives are the *trace*, the *stable-failures*, and the *failures-divergences* semantics. The former gives an abstract view to a process, because only sequences of events are considered, and the failures-divergences semantics gives the most accurate view, also including all deadlocks and divergences of a process. As we are basically interested in safety properties here, the trace semantics suffices for our needs.

Traces can easily be extracted from transition systems forming the operational semantics of CSP. It is defined as a *labelled transition system (LTS)*

$$(Q, \mathit{Events}^{\tau\checkmark}, q_0, \longrightarrow),$$

with a special event $\checkmark$ representing the termination of a process, and an internal event τ that is not visible from outside. $Events^{\tau}$ is the set of events including τ, $Events^{\checkmark}$ the set of events including $\checkmark$, and $Events^{\tau\checkmark}$ the combination of both. The set Q contains all CSP processes, and $q_0 \in Q$ is the initial process. The transition relation $\longrightarrow$ is defined inductively over the structure of CSP processes by the following firing-rules.

$$\frac{}{\texttt{Skip} \xrightarrow{\checkmark} \Omega} \qquad \text{(skip)}$$

$$\frac{}{a \rightarrow P \xrightarrow{a} P} \qquad \text{(prefix)}$$

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'} \qquad \frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'} \qquad a \in Events^{\checkmark} \qquad \text{(extchoice)}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q} \qquad \text{(intchoice)}$$

$$\frac{P \xrightarrow{b} P'}{P \underset{A}{\|} Q \xrightarrow{b} P' \underset{A}{\|} Q} \qquad \frac{Q \xrightarrow{b} Q'}{P \underset{A}{\|} Q \xrightarrow{b} P \underset{A}{\|} Q'} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{A}{\|} Q \xrightarrow{a} P' \underset{A}{\|} Q'} \qquad \frac{}{\Omega \underset{A}{\|} \Omega \xrightarrow{\checkmark} \Omega} \qquad \begin{array}{l} b \in Events^{\tau\checkmark} \setminus A \\ a \in A \setminus \{\tau\} \end{array} \qquad \text{(parallel)}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{a} P'}{P \,;\, Q \xrightarrow{a} P' \,;\, Q} \qquad a \in Events^{\tau} \qquad \text{(seqcomp)}$$

$$\frac{P \xrightarrow{\tau} P'}{P[R] \xrightarrow{\tau} P'[R]} \qquad \frac{P \xrightarrow{\checkmark} P'}{P[R] \xrightarrow{\checkmark} \Omega} \qquad \frac{P \xrightarrow{a} P'}{P[R] \xrightarrow{b} P'[R]} \qquad aRb \qquad \text{(renaming)}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{a} P' \setminus B} \qquad \frac{P \xrightarrow{b} P'}{P \setminus B \xrightarrow{\tau} P' \setminus B} \qquad a \notin B, b \in B \qquad \text{(hiding)}$$

$$\frac{}{X \xrightarrow{\tau} P} \qquad X \stackrel{c}{=} P \qquad \text{(call)}$$

$$\frac{}{\mu X \bullet F(X) \xrightarrow{\tau} F(\mu X \bullet F(X))} \qquad \text{(recursion)}$$

We additionally use the convention that $\Omega \setminus A = \Omega$ and $\Omega[R] = \Omega$.

We extend this single-step transition relation to a multi-step relation: The τ-step transition relation $P \stackrel{a}{\Longrightarrow} \overline{P}$ represents the τ-closure of the single-step relation $\xrightarrow{a}$, i.e., $P \stackrel{a}{\Longrightarrow} \overline{P}$ means that the LTS of P has a run from P to $\overline{P}$

$$\langle a_0, a_1, \ldots, a_i, a, a_{i+1}, \ldots, a_n \rangle, \tag{2.1}$$

such that $a_i = \tau$ holds for all $i \in 0..n$. The run (2.1) exists if there are transition steps $P_i \xrightarrow{a_i} P_{i+1}$ for $i \in 0..n$ and $P_0 = P, P_{n+1} = \overline{P}$. Furthermore, for a sequence of events $w = \langle a_0, \ldots, a_n \rangle$ the multi-step transition relation $P \stackrel{w}{\Longrightarrow} \overline{P}$ denotes the consecutive application of the single-step relation

$$P \stackrel{w}{\Longrightarrow} \overline{P} \quad := \quad P \stackrel{a_0}{\Longrightarrow} \circ \cdots \circ \stackrel{a_n}{\Longrightarrow} \overline{P}.$$

We use $\Longrightarrow^*$ to denote a sequence of arbitrary transition steps, i.e., $P \stackrel{w}{\Longrightarrow} \overline{P}$ with arbitrary w.

With this multi-step transition relation, we can define the set of finite, prefix-closed traces of a process P by

$$\mathit{traces}(P) := \{w \mid \exists\, Q : P \stackrel{w}{\Longrightarrow} Q\}.$$

Even though the systems we consider are reactive and usually have infinite behaviour, it is sufficient to include finite traces in the semantics, because the infinite traces can be computed from the set of all finite traces. This holds as long as no infinite branching processes are allowed [Ros98], which we explicitly exclude here.

2.1.2. Object-Z

Object-Z [Smi00] is an object-oriented extension for the mathematical Z notation that has been developed mainly in the 1980s [Spi92]. It was standardised in 2002 [ISO02]. Z provides a notation for logical operators, sets, relations, and, based on this, a *mathematical toolkit* containing Z definitions for common mathematical operators and expressions like $\cup, \cap, \subseteq, \varnothing$, and notation for natural numbers and integers as well as sequences. One of the most obvious features of the Z notion is the use of a specific schema notation to structure a specification into smaller pieces. To give an example, the schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Train \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
maxbd, maxcd, maxspd, ma, pos : \mathbb{R} \\
\hline
maxbd > 0 \\
maxcd > 0 \\
maxspd > 0 \\
\hline
\end{array}$$

describes (a part of) the state space of a simple train control system: it declares three real-valued variables that need all to be larger than 0 and the variables *ma* and *pos*. The lines below the horizontal dash contain a predicate that has to be valid for the declared variables (the predicates in each line are connected by conjunction). Such a schema defines the set of all mappings from names (*maxbd*, *maxcd*, *maxspd*, *ma*, and *pos*) to values from the corresponding domain ($\mathbb{R}$). These mappings are called *bindings*, written

$$\langle\!|\ maxbd == 3.2, maxcd == 2.5, maxspd == 500, ma == 1000, pos == -1.7\ |\!\rangle,$$

which is a valid binding for the schema above. Schemas can also be written in a condensed form, e.g., the *Train* schema can also be defined by

$$Train == \\ \quad [maxbd, maxcd, maxspd, ma, pos : \mathbb{R} \mid maxbd > 0 \land maxcd > 0 \land maxspd > 0].$$

Variables are also used in *decorated* form with the standard decorations $'$, !, and ?. A *primed* variable x' denotes the state of the variable x after an operation, while ! and ? distinguish output and input variables, respectively. For instance, the operation schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad extend \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta(Train) \\
newMA? : \mathbb{R} \\
\hline
ma' = newMA? \\
\hline
\end{array}$$

declares an input variable *newMA*? of type real and sets the new value of variable *ma* to the value of this input variable. The Δ-expression in the first line introduces the symbols from the schema *Train*. Additionally, primed versions of all symbols of the *Train* schema are introduced such that ma' is actually declared for the *extend* operation.

Z Syntax

The basic building blocks in Z are paragraphs comprising the already introduced schemas, axiomatic descriptions, basic type definitions, abbreviation definitions, and

free type definitions:

$Z_Paragraph$ ::=	─ $NAME$ ─── │ $DeclPart$ [├─── │ $Predicate$] └───	(schema)
|	│ $DeclPart$ [├─── │ $Predicate$]	(axiomatic description)
	| $[NAME, \ldots, NAME]$	(basic type definition)
	| $NAME == Expression$	(abbreviation definition)
	| $NAME$ '::=' $Branch$ '|' ... '|' $Branch$	(free type definition)

$$Branch ::= NAME \mid NAME \ll Expression \gg$$

Axiomatic descriptions introduce possibly constrained global variables, and basic type definitions allow for introducing identifiers for new types. Moreover, abbreviations can be defined, e.g., $Position == \mathbb{R}^+$ declares a new type *Position*. Finally, free types can be declared in a BNF syntax.

DeclPart is a declaration of the shape $NAME_1 : TYPE_1, \ldots, NAME_n : TYPE_n$. A *Predicate* is a Z expression that evaluates to true or false and comprises the standard operators from predicate logic, $\wedge, \vee, \Rightarrow, \Leftrightarrow$ and quantifications

$$\begin{aligned} Predicate \quad &::= \forall\, SchemaText \bullet Predicate \mid \\ &\quad\;\; \exists\, SchemaText \bullet Predicate \\ SchemaText &::= DeclName\ [\ `|\text{'}\ Predicate\ ]\ . \end{aligned}$$

For instance, monotonicity of a function $f : \mathbb{R} \to \mathbb{R}$ is defined in Z syntax by

$$\forall\, x, y : \mathbb{R} \mid x \leq y \bullet f(x) \leq f(y).$$

To avoid unnecessary mixing of syntactic constructs, we will use the notation of Z for predicates throughout this document.

Remark 2.1.2 (Booleans and reals in Z). Note that we follow the approach of Hoenicke with respect to the Boolean type $\mathbb{B}$. The type $\mathbb{B}$ is often useful but not directly declared in the Z standard. Nevertheless, as explicated in [Hoe06], $\mathbb{B}$ can easily be defined as abbreviation in standard Z syntax, so we consider $\mathbb{B}$ as given. Analogously, the type $\mathbb{R}$ is also not defined in the mathematical toolkit of the Z standard. We do not explicitly give an axiomatisation of real numbers in terms of a Z specification and instead refer to existing works on this [Art96, OB97, Toy98].

Z Semantics

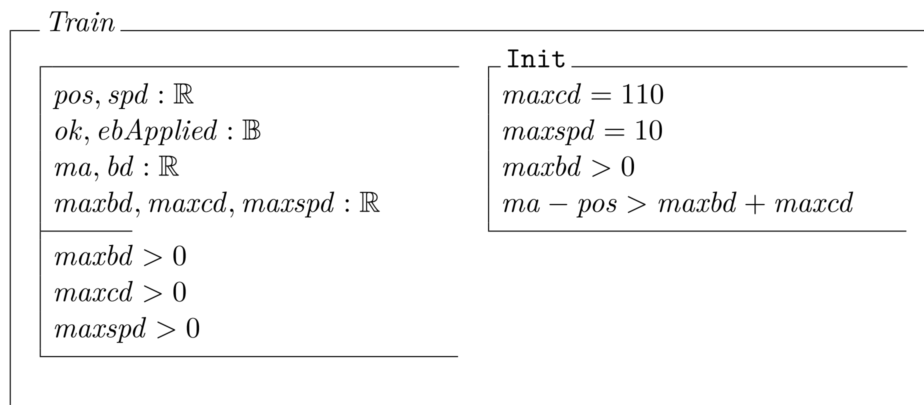
The Z ISO standard [ISO02] defines the semantics of Z expressions as the set of possible values for all named symbols. The following symbols are used to describe the Z semantics: $\mathbb{U}$ is the universe of all semantical values for Z expressions, $\mathbb{W}$ is a set of subsets of $\mathbb{U}$ for expressions that are not generic, and $Model : NAME \ffun \mathbb{U}$ assigns values to symbols, where $NAME$ is the set of all possible named Z symbols, and $\ffun$ denotes a finite partial function. With these basic elements, the semantical relations on Z terms are defined:

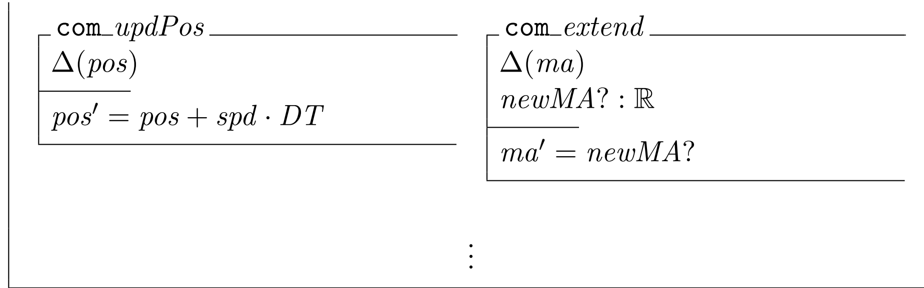
- $[\![Paragraph]\!]^{\mathcal{D}} \in Model \leftrightarrow Model$ assigns a meaning to paragraphs with the intuition that a paragraph declaring or restricting Z symbols relates an input model to a model that is extended according to the paragraph.
- $[\![Predicate]\!]^{\mathcal{P}} \in \mathbb{P} Model$ assigns a set of valid models to a predicate.
- $[\![Expression]\!]^{\mathcal{E}} \in Model \rightarrow \mathbb{W}$ assigns a meaning to Z expressions, i.e., for a given model of the symbols from the expression, the expression is evaluated to a value from the semantical domain $\mathbb{W}$.

Object-Z

Object-Z (OZ) has been developed by Graeme Smith [Smi92, Smi00] to facilitate object-oriented modelling with a Z-based language. Thus, it incorporates the standard concepts of object-orientation like classes, objects, inheritance, and polymorphism. Beyond that, OZ is a conservative extension of Z such that all Z specifications are also valid OZ specifications.

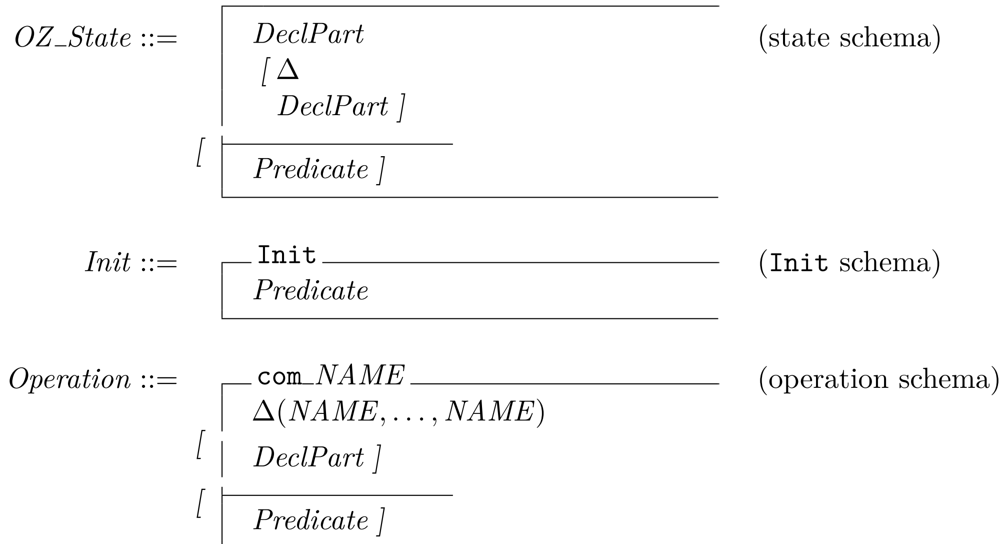
To give an example of the main OZ construct, a class specification, we extend the *Train* example from the beginning of this section:





This class *Train* (which is not completely presented here) describes the state space of *Train* objects together with possible operations on the state space, as well as all allowed initial values of attributes. The state space is defined with a so-called *state schema*, that defines, similarly to a standard Z schema, real-valued variables *pos* and *spd* representing the position and speed values of a train, some Boolean attributes *ok*, *ebApplied* etc. The schema marked with `Init` contains a predicate, defining initial values of the class attributes. The named schemas `com`_*updPos* and `com`_*extend* are the operations of the class, which change the state space. The use of such operations is the only way to change the state space of the class. For example, the operation `com`_*updPos* defines how the position variable *pos* is updated (DT is the length of the time interval from the last position update). Analogously to standard Z schemas, the Δ-expression is used to introduce the primed version of *pos*, i.e., a (primary) variable can only be changed if it occurs in the Δ-list of an operation. The `com`_*extend* schema additionally declares an input variable *newMA*?.

The syntax of state schema, `Init` schema, and operation schema is defined by:



2.1.3. Duration Calculus

The Duration Calculus (DC) is an interval based dense real-time logic that was introduced in the early 1990s by Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn [ZHR91] as part of the ProCoS project [HHF$^+$94]. With DC formulae, properties over the duration of states in given time intervals can be formulated. The textbook [ZH04] gives a general overview on theoretical results on the DC. The following sections explicate syntax and semantics of DC.

DC Symbols

The syntax of DC is built on the following set of symbols.

- *Observables* $X \in \mathit{Obs}$. Observables (or *state variables*) are time-dependent variables having a finite domain, denoted D_X for an observable X. We use continuous time as time domain: $\mathit{Time} == \mathbb{R}^+$ (for DC with discrete time see, e.g., [ZH04]).
- Functions $f : \mathbb{R}^n \to \mathbb{R}$ and predicate symbols $p : \mathbb{R}^m \to \mathbb{B}$.
- Time-independent variables Vars.

These symbols are interpreted by the following mappings:

- The semantics of an observable X is a mapping from time into the data domain of the observable

  $$\mathcal{I}[\![X]\!] : \mathit{Time} \to D_X,$$

  called *interpretation*. The mapping $\mathcal{I}[\![X]\!]$ is required to have finite variability, i.e., in every finite time-interval a state is not changed infinitely often.
- Function and predicate symbols are interpreted by corresponding functions

  $$\widetilde{f} : \mathbb{R}^n \to \mathbb{R} \quad \text{and} \quad \widetilde{p} : \mathbb{R}^m \to \mathbb{B}.$$

  We always interprete the arithmetical function symbols $+, -, *, /$ and relations $<, \leq, =, \geq, >$ by their standard meaning.
- Global variables are interpreted by valuations

  $$\mathcal{V}(x) \in \mathbb{R}.$$

  The set of all valuations is denoted by Val.

State Assertions

State assertions (also *state expressions*) describe the state of the system at a given point in time, and they are defined by the following grammar:

$$\mathit{StateExpr} ::= 0 \mid 1 \mid X = d \mid \neg \mathit{StateExpr} \mid (\mathit{StateExpr} \land \mathit{StateExpr}),$$

where $d \in D_X$. The values 0 and 1 represent the Boolean constants true and false (for the purpose of the definitions of this section, we identify $\mathbb{B}$ with the set $\{0, 1\}$). The remaining Boolean connectives like $\lor, \Rightarrow$ are considered as abbreviations.

The semantics of state assertions, which are also time-dependent, is given by the inductively defined extension of interpretations $\mathcal{I}$ from observables to state assertions, $\mathcal{I}[\![\mathit{StateExpr}]\!] : \mathit{Time} \to \mathbb{B}$.

$$\begin{aligned}
\mathcal{I}[\![0]\!](t) &= 0 \\
\mathcal{I}[\![1]\!](t) &= 1 \\
\mathcal{I}[\![X = d]\!](t) &= \begin{cases} 1, & \text{if } \mathcal{I}[\![X]\!](t) = d \\ 0, & \text{otherwise} \end{cases} \\
\mathcal{I}[\![\neg\pi]\!](t) &= 1 - \mathcal{I}[\![\pi]\!](t) \\
\mathcal{I}[\![\pi_1 \land \pi_2]\!](t) &= \begin{cases} 1, & \text{if } \mathcal{I}[\![\pi_1]\!](t) = 1 \text{ and } \mathcal{I}[\![\pi_2]\!](t) = 1 \\ 0, & \text{otherwise.} \end{cases}
\end{aligned}$$

DC Terms

DC terms express the duration of states in fixed time-intervals. The syntax is defined by

$$\mathit{Term}_{DC} ::= x \mid \ell \mid \int \mathit{StateExpr} \mid f(\mathit{Term}_{DC}, \ldots, \mathit{Term}_{DC}),$$

where x is a global variable, f a function symbol, and ℓ a special symbol referring to the length of a time-interval.

A term θ is interpreted by a mapping

$$\mathcal{I}[\![\theta]\!] : \mathit{Val} \times \mathit{Intv} \to \mathbb{R}$$

over a fixed interval from the set

$$\mathit{Intv} := \{[b, e] \mid b, e \in \mathit{Time}, b \leq e\}$$

and a valuation of global variables from Val. The mapping is inductively defined by

$$\begin{aligned}
\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) &= \mathcal{V}(x) \\
\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) &= e - b \\
\mathcal{I}[\![\int \pi]\!](\mathcal{V}, [b, e]) &= \int_b^e \mathcal{I}[\![\pi]\!](t)\, dt \\
\mathcal{I}[\![f(\theta_1, \ldots, \theta_n)]\!](\mathcal{V}, [b, e]) &= \widetilde{f}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e])),
\end{aligned}$$

with terms θ_i and a state assertion π. That is,the semantics of ℓ is the length of the considered interval. The semantics of a term $\int \pi$ is the overall time, in which π is evaluated to true on the considered interval. The expression $\int_b^e \mathcal{I}[\![\pi]\!](t)\,dt$ is Riemann-integrable because of the finite-variability of all observables. Therefore, the definition is well-defined.

If a term does not contain global variables, we omit $\mathcal{V}$ and abbreviatory write $\mathcal{I}[\![\mathit{Term}_{DC}]\!][b,e]$ instead.

DC Formulae

Based on DC terms, we now define DC formulae:

$$\begin{aligned}\mathit{Formula}_{DC} ::= {} & p(\mathit{Term}_{DC}, \ldots, \mathit{Term}_{DC}) \mid \neg \mathit{Formula}_{DC} \\ & \mid (\mathit{Formula}_{DC} \wedge \mathit{Formula}_{DC}) \mid \forall\, x : \mathit{Formula}_{DC} \mid \mathit{Formula}_{DC} {}^\frown \mathit{Formula}_{DC}.\end{aligned}$$

Besides the usual Boolean connectives and quantifications (the missing connectives are again defined as abbreviations), DC formulae may contain the essential *chop* operator $F {}^\frown G$, dividing an interval into two sub-intervals such that F holds on the first sub-interval and G on the second.

The semantics of a DC formula F is defined by the mapping

$$\mathcal{I}[\![F]\!] : \mathit{Val} \times \mathit{Intv} \to \mathbb{B}$$

with

$$\begin{array}{lll}\mathcal{I}[\![p(\theta_1, \ldots, \theta_n)]\!](\mathcal{V}, [b,e]) & \text{iff} & \widetilde{p}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b,e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b,e])) \\ \mathcal{I}[\![F_1 \wedge F_2]\!](\mathcal{V}, [b,e]) & \text{iff} & \mathcal{I}[\![F_1]\!](\mathcal{V}, [b,e]) \text{ and } \mathcal{I}[\![F_2]\!](\mathcal{V}, [b,e]) \\ \mathcal{I}[\![\forall\, x : F_1]\!](\mathcal{V}, [b,e]) & \text{iff} & \text{for all } d \in \mathbb{R} \text{ holds } \mathcal{I}[\![F_1]\!](\mathcal{V}[x := d], [b,e]) \\ \mathcal{I}[\![F_1;\ F_2]\!](\mathcal{V}, [b,e]) & \text{iff} & \text{there is } m \in [b,e] \text{ such that} \\ & & \mathcal{I}[\![F_1]\!](\mathcal{V}, [b,m]) \text{ and } \mathcal{I}[\![F_2]\!](\mathcal{V}, [m,e]),\end{array}$$

where the F_i are DC formulae and the θ_i DC terms. We again omit the valuation $\mathcal{V}$ if it is not needed, and we say that a DC formula F *holds on an interval* $[b,e]$ *for* $\mathcal{I}$ (written $\mathcal{I}, [b,e] \models F$) iff $\mathcal{I}[\![F]\!](\mathcal{V}, [b,e])$ holds for all valuations $\mathcal{V}$. We say that $\mathcal{I}$ *satisfies* F (written $\mathcal{I} \models F$) iff F holds for an interval $[0,t]$ for $\mathcal{I}$:

$$\mathcal{I}, [0,t] \models F \text{ for a } t \in \mathit{Time} \text{ with } t > 0.$$

We sometimes write $[\![F]\!]$ if we mean the set of all interpretations satisfying F.

The following useful syntactic constructs are defined as abbreviations:

$$\lceil\rceil := \ell = 0 \qquad\qquad (\textit{point interval})$$

$$\lceil \varphi \rceil := (\int \varphi = \ell) \wedge (\ell > 0) \qquad (\varphi \text{ holds everywhere})$$
$$\Diamond F := \text{true} \frown F \frown \text{true} \qquad (F \text{ holds eventually})$$
$$\Box F := \neg \Diamond \neg F \qquad (F \text{ holds always}).$$

Integrating Z Expressions into DC

Up to now, we considered standard DC with state assertions over observables with a finite domain. But we are interested in a uniform integration of DC with CSP and OZ, where Z predicates are used to describe the state of a system. Hence, it is preferable to use Z predicates for definitions of system states in DC formulae. Hence, we extend the syntax for DC state assertions by Z predicates:

$$\mathit{StateExpr} ::= \mathit{Predicate}.$$

In Z, the semantics of a predicate φ is given by $[\![\varphi]\!]^{\mathcal{P}}$, i.e., a set of models. Thus, to interprete the predicate as a DC state assertion, which needs to be time-dependent, we use—analogously to interpretations for observables—a mapping assigning a model to the symbols in φ for every point in time: $\mathcal{I}_Z : \mathit{Time} \rightarrow \mathit{Model}$. We again require the finite-variability for $\mathcal{I}$. Then, a state assertion φ can be interpreted by

$$\mathcal{I}[\![\varphi]\!](t) = \begin{cases} 1, & \text{if } \mathcal{I}_Z(t) \in [\![\varphi]\!]^{\mathcal{P}} \\ 0, & \text{otherwise.} \end{cases}$$

Event Model

The semantical domain of DC and Z symbols is defined in terms of models or valuations if a symbol is interpreted at a given point in time. It is defined in terms of interpretations if a symbol is interpreted over a time-interval. In order to integrate events into DC (and later into CSP-OZ-DC) it is therefore helpful to express the event notion of CSP in the same way. Thus, we model events with Boolean variables with the idea that the corresponding event occurs whenever the Boolean variable changes its value.

This notion of events can be integrated into the DC. The occurrence of an event e is written as $\updownarrow e$ and defined by

$$\mathcal{I}, [b, e] \models \updownarrow e \text{ iff } b = e \wedge \exists\, n, m \in \mathit{Time} \mid n < b < m \bullet \mathcal{I}[\![e]\!]([n, b]) \neq \mathcal{I}[\![e]\!]([e, m]).$$

With this, we can also define the non-occurrence of events in point intervals ($\not\updownarrow e$) and on non-singular intervals ($\boxminus e$):

$$\not\updownarrow e == \neg \updownarrow e \wedge \ell = 0$$
$$\boxminus e == \neg(\ell > 0 \frown \updownarrow e \frown \ell > 0).$$

DC Counterexample Traces

The DC is an expressive language to formulate dense real-time properties, but it is undecidable [ZH04] and cannot be implemented in general. A standard example substantiating this issue is the DC formula

$$\neg\Diamond(\updownarrow a ^\frown \ell = 1 ^\frown \updownarrow a), \tag{2.2}$$

which requires that exactly one time unit after an event a no a event occurs. This formula cannot be implemented with a finite number of clocks, because we need a new clock for every a event that occurs.

For this reason, there has been a lot of work in identifying sub-sets of DC that are suitable for automated verification [Rav94, BLR95, Pan02, Frä04, Hoe06, FH07, MFHR08]. In particular, Hoenicke defines DC counterexample traces that can be translated into Phase Event Automata and, thus, are suitable for the use with CSP-OZ-DC. Their syntax is defined as follows:

$$\begin{aligned}
\mathit{ce_formula} &::= \neg(\mathit{phase} ^\frown (\mathit{phase} \mid \mathit{events}) \\
&\quad\; ^\frown \cdots ^\frown (\mathit{phase} \mid \mathit{events}) ^\frown \mathrm{true}) \\
\mathit{phase} &::= (\mathrm{true} \mid \lceil \mathit{Predicate} \rceil)\ [\ \wedge \ell \sim t \mid \wedge\, \ell \sim \mathit{NAME}\ ] \\
&\quad\; [\ \wedge \boxminus \mathit{NAME} \cdots \wedge \boxminus \mathit{NAME}\ ] \\
\sim &::= \ \leq \ \mid \ < \ \mid \ > \ \mid \ \geq \\
\mathit{events} &::= \updownarrow \mathit{NAME} \mid \not\updownarrow \mathit{NAME} \\
&\quad\; \mid \mathit{events} \wedge \mathit{events} \mid \mathit{events} \vee \mathit{events}.
\end{aligned}$$

By this, one cannot specify phases with an exact length; thus, problematic formulae like (2.2) are excluded. Following [FJSS07], this definition slightly extends Hoenicke's counterexample formulae, because symbolic constants can be used to specify the length of intervals: $\ell \sim \mathit{NAME}$.

2.2. Combining CSP, OZ, and DC Into a Parametric Specification Language

We now integrate the three single languages of the previous section into the combined formalism CSP-OZ-DC. The definitions of syntax and semantics of CSP-OZ-DC basically follow the definitions in [Hoe06], but are slightly widened with respect to parametric systems. Generally, we are interested in several notions of parameters:

- Data parameters
- Timing parameters
- Parametric compound processes

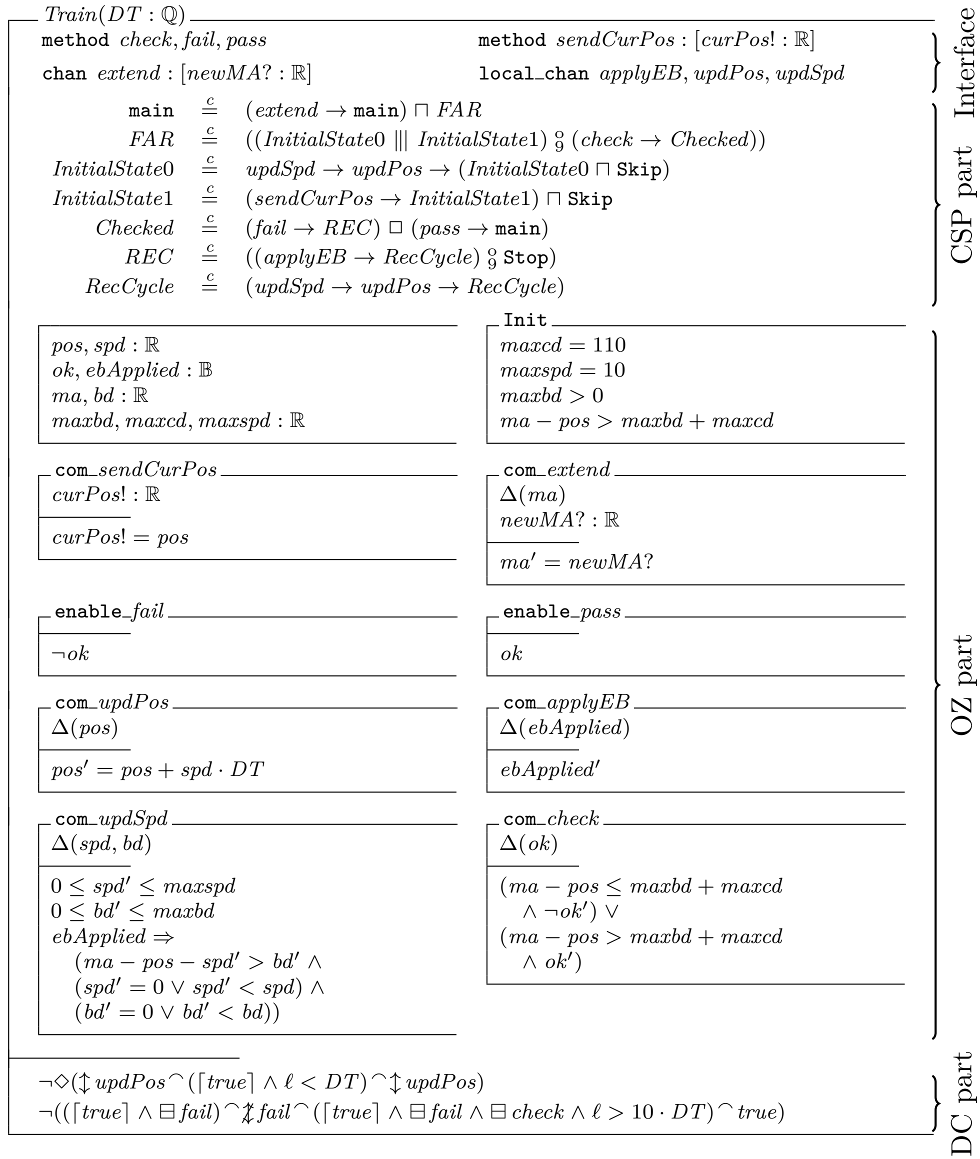


Figure 2.1.: Exemplary CSP-OZ-DC specification

- Parametric number of parallel processes

The good news is that the standard definition of CSP-OZ-DC already supports parametric specifications: data parameters—specified with Z—are inherently integrated in the of CSP-OZ-DC. Timing parameters of CSP-OZ-DC are introduced in [FJSS07]. Nevertheless, CSP-OZ-DC does not directly support compound processes. We will develop a notion of parametric compound processes in Chap. 3 up to Chap. 6 that

is used in our Verification Architecture approach and that integrates well with the CSP-OZ-DC approach. Verification of a parametric number of processes is not part of this work.

2.2.1. Syntax of CSP-OZ-DC

A *(parametric) CSP-OZ-DC* specification is defined as follows.

$$
\mathit{Param_COD_Class} ::= \begin{array}{|l}
\mathit{NAME}\ [\,[\mathit{NAME}, \ldots, \mathit{NAME}]\,]\ \ [\,(\mathit{SchemaText})\,] \\
\mathit{Interface} \\
\mathit{Paragraphs} \\
\mathit{OZ_State} \\
\mathit{Init} \\
\ [\ \mathit{Operation} \cdots \mathit{Operation}\ ] \\
\hline
[\ \mathit{DC}\ ] \\
\hline
\end{array}
$$

$$
\begin{aligned}
\mathit{Paragraphs} ::= {} & \mathit{ProcessDeclaration}\ \mathit{NL} \\
& [\ \mathit{Z_Paragraph} \cdots \mathit{Z_Paragraph}\ ]
\end{aligned}
$$

NL stands for a line break. So, a CSP-OZ-DC extends the OZ concept of a class by an interface, a process declaration part, and the DC part, which are introduced below. The remaining elements, *Z_Paragraph*, *OZ_State*, *Init*, and *Operation*, are the same as for OZ classes as defined in Sect. 2.1.2. Behind the class name, a class may introduce formal parameters: in square brackets a list of free types can be defined that are local to the class, optionally followed by round brackets declaring further class parameters. For instance, the class *Train* in Fig. 2.1—that gives an exemplary CSP-OZ-DC specification for the train controller of our running example—has a rational parameter DT, which is used as time constant to model the cycle time of a train.

The basic idea for the integration of OZ and CSP is that the CSP part structures the control flow of the operations occurring in the OZ part. At every point in time when a CSP event occurs, the corresponding state change from the OZ part is executed. The DC formulae can further restrict the control flow by imposing timing constraints on the events. With the embedding of Z expressions into DC formulae from Sect. 2.1.3, the DC part can additionally restrict the state changes from the OZ part.

Interface

At the beginning of every CSP-OZ-DC class the interface to other classes needs to be declared. The interaction of CSP-OZ-DC classes is modelled with CSP processes, which fully encapsulates the inter-class communication in CSP-OZ-DC. Thus, the interface needs to declare all CSP events or channels that can be used for synchron-

isations. The syntax of the interface part is defined by:

$$\begin{aligned} \mathit{Interface} &::= \mathit{ChannelDecl}\ \mathit{NL} \ldots \mathit{NL}\ \mathit{ChannelDecl} \\ \mathit{ChannelDecl} &::= \texttt{chan}\mathit{NAME}, \ldots, \mathit{NAME}\ [\ :[\mathit{DeclPart}]\ ] \\ &\quad \mid \texttt{method}\mathit{NAME}, \ldots, \mathit{NAME}\ [\ :[\mathit{DeclPart}]\ ] \\ &\quad \mid \texttt{local_chan}\mathit{NAME}, \ldots, \mathit{NAME}\ [\ :[\mathit{DeclPart}]\ ] \end{aligned}$$

There are three types of channels, global channels, local channels, and methods. The channel types `chan` and `method` only differ in a software engineering view: `method` is used when it is intended that the class implements itself, whereas `chan` indicates that the method from another class is called. Formally there is no difference. Local channels are not visible to the outside of the class and cannot be used for synchronisations. Channels can have parameters whose types are specified with a *DeclPart* declaration. These parameters are used to share data values between classes.

The example in Fig. 2.1 declares the methods *check*, *fail*, and *pass*, for which operations are defined in the OZ part. The control flow of the methods is restricted by the process equation in the CSP part (which is identical to the CSP equation system from Example 2.1.1). The methods *sendCurPos* and the channel *extend* have real-valued parameters for sending the current train position (output variable *curPos*!) to an environment component (for instance the RBC) and for receiving a new movement authority (input variable *newMA*?).

Process Declarations

The CSP part in CSP-OZ-DC consists of a process equation system with standard CSP processes (Sect. 2.1.1) extended with unknown processes in order to deal with the structures we develop in this work.

$$\begin{aligned} \mathit{ProcessDeclaration} &::= \mathit{ProcessEquationNL} \cdots \mathit{NL}\ \mathit{ProcessEquation} \\ \mathit{ProcessEquation} &::= \mathit{NAME} \overset{c}{=} (\mathit{Process} \mid \mathit{UnknownProc}) \\ \mathit{UnknownProc} &::= \texttt{Proc}_{\setminus A,V}, \end{aligned}$$

where $A \subseteq \mathit{Events}$ and $V \subseteq \mathit{State}$. A special process `main` is used to indicate the process that is initially active. The idea of an unknown process $\texttt{Proc}_{\setminus A,V}$ is that arbitrary behaviour is allowed expect for events from the exclusion alphabet A and except state changes in V. Unknown processes are analysed in the following chapters.

To simplify the presentation, we use a restricted version of CSP processes in CSP-OZ-DC specifications here. Fischer and Hoenicke additionally allow Z schema expressions within the CSP part (e.g., to pass local values from one event to another or to parameterise process equations) and also define parameterised operators for parallel composition and choice. We go without these more complex CSP expressions, because they are not supported by the verification tools we use for our case studies. For a full definition of *ProcessDeclaration* see [Fis00, Hoe06].

DC Part

The DC part consists of a list of DC counterexample traces.

$$DC ::= ce_formula\ NL \dots NL\ ce_formula$$

Events and variables of the DC part need to be defined in the interface or in the state schema of the class.

The example of Fig. 2.1 contains two counterexample formulae. The first one specifies that there are at least DT time units between two *updPos* events, where DT has been defined as class parameter. The second states that as long as no *fail* event occurs, a *check* event occurs at least every $10 \cdot DT$ time units. A second possibility would have been to enforce a similar behaviour by accessing the state space of the class, e.g., with a formula

$$\neg(\lceil ok \rceil ^\frown (\lceil ok \rceil \wedge \boxminus\, check \wedge \ell > 10 \cdot DT) ^\frown true),$$

demanding the desired behaviour for *check* as long as the Boolean variable *ok* has the value true.

Note that the state space of suchlike systems is infinite due to several reasons: we have a system with dense real-time behaviour, the state space is infinite because of infinite data types like reals or possibly rich infinite data structures like lists, and processes with unknown parts represent an infinite number of concrete processes. Moreover, the case study requires the transfer of messages including data values from infinite domains (real-valued train positions), which is not possible within the standard CSP approach, where message transfer is realised via compound events [Sch99].

2.2.2. Semantics of CSP-OZ-DC

We have already defined the semantics of DC formulae with events and data in terms of interpretations, which is a suitable semantical domain also for the CSP-OZ-DC combination. Overall, the possible interpretations of a CSP-OZ-DC specification can be directly computed by the interpretations of its single parts.

So far the semantics of a CSP process is a set of traces of events, but we have already expressed events by changes of Boolean variables. Therefore, traces can canonically be modelled by interpretations. An interpretation $\mathcal{I}$ satisfies a process P, written $\mathcal{I} \models P$, iff there is a run $\langle a_1, a_2, \dots \rangle$ of the LTS of P and points in time $t_0, t_1, \dots \in Time$ with $0 = t_0 < t_1 < t_2 < \dots$ and models $\mathcal{I}(t) = \mathcal{M}_i$ for $t_i \leq t < t_{i+1}$ such that

$$\mathcal{M}_i(a_{i+1}) \neq \mathcal{M}_{i+1}(a_{i+1}) \text{ for } i \geq 0,$$

where the a_i are Boolean variables. The sequence $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$ is called *untimed sequence of* $\mathcal{I}$ and denoted with $untime(\mathcal{I})$. If the `main` process is satisfied by an interpretation, we also write $\mathcal{I} \models CSP_Part$.

The semantics of Z expressions has been given by models of its symbols. The semantics of OZ is in [Smi92] defined by the history of models that an object has passed through, which basically are untimed interpretations. We say that an interpretation $\mathcal{I}$ with $untime(\mathcal{I}) = \langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$ satisfies the OZ part of a class, $\mathcal{I} \models OZ_part$, iff

$$\begin{aligned}
&\mathcal{M}_0 \in [\![Init]\!]^{\mathcal{P}}\\
&\mathcal{M}_i \in [\![State]\!]^{\mathcal{P}}\\
&\mathcal{M}_{i-1} \cup \mathcal{M}'_i \cup \mathcal{M}_i(op_i\bowtie) \in [\![\mathtt{com}_op_i]\!]^{\mathcal{P}}
\end{aligned}$$

for an arbitrary sequence of operation schemas $\mathtt{com}_op_i$. $\mathcal{M}'_i$ is a valuation of primed symbols: $\mathcal{M}'_i(v') = \mathcal{M}_i(v)$. The symbol $\bowtie$ is used to distinguish parameter variables of operations; $op_i\bowtie$ represents the parameters of the operation op_i (for details on handling of operation parameters see [Hoe06]).

With these arrangements we can define the semantics of CSP-OZ-DC specifications.

Definition 2.2.1 (Semantics of CSP-OZ-DC specifications)
We define the semantics of a CSP-OZ-DC specification as a set of interpretations $\mathcal{I} : Time \to Model$:

$$\left[\!\!\left[\begin{array}{l} [\,Interface \\ \;CSP_part \\ \;Paragraphs \\ \;OZ_part \\ \;\mid DC_part] \end{array}\right]\!\!\right]^{\mathcal{E}} \mathcal{M} \;:=\; \begin{array}{l} \{\mathcal{M}, \mathcal{M}', \mathcal{M}'' : Model;\ \mathcal{I} : Time \to Model \mid \\ \;\mathcal{N} \in [\![Interface]\!]^{\mathcal{E}}\mathcal{M} \\ \;\wedge (\mathcal{M} \cup \mathcal{N}, \mathcal{M}') \in [\![Paragraphs]\!]^{\mathcal{D}} \\ \;\wedge \forall\, t : Time \bullet \mathcal{M}' \cup \mathcal{M}'' \subseteq \mathcal{I}(t) \\ \;\wedge\, \mathcal{I} \models CSP_part \\ \;\wedge\, \mathcal{I} \models OZ_part \\ \;\wedge\, \mathcal{I} \models DC_part \\ \;\bullet\, \mathcal{I}\ \}. \end{array}$$

The model $\mathcal{M}$ is the model of the environment of the class. The interface is defined as a set of variable constructions, and thus, its semantics is, according to the Z standard [ISO02], given by the set of bindings $[\![Interface]\!]^{\mathcal{E}}\mathcal{M}$, declaring the channel types. Therefore, in the CSP-OZ-DC semantics we need to extend the model of the environment $\mathcal{M}$ by these bindings for the channels and by all local declarations in *Paragraphs*. Due to $\mathcal{I} \models OZ_part$, the state variables are defined in the models $\mathcal{I}(t)$, and the same holds for the events from *CSP_part*.

In the original definition of the CSP-OZ-DC semantics in [Hoe06], only interpretations over public channels are included by which only these channels are visible outside of classes. We here use a *visibility semantics*, in which the full interpretations of all symbols are included, because the old semantics suffers from the fact that the state variables are not visible in the semantics such that, formally, properties over state variables cannot directly be verified.

Sometimes it is necessary to access all possible initial models of a CSP-OZ-DC specification. We therefore define

$$Init(cod) \;:=\; [\![Init]\!]^{\mathcal{P}} \cap [\![State]\!]^{\mathcal{P}} \cap \bigcup_{\mathcal{M}:Models} [\![Paragraphs]\!]^{\mathcal{D}}\mathcal{M}.$$

So, $Init(cod)$ contains all models that satisfy the state schema and the initial schema of *cod* and that respect the axiomatic declarations of the paragraph section.

We also define the semantics of systems of specifications. As we use the visibility semantics for CSP-OZ-DC, we define the parallel composition of CSP-OZ-DC specifications only for classes with disjoint symbols except for public channels. If this is not the case then local variables like state variables or local constants and channels have to be renamed to unique, new identifiers.

Definition 2.2.2 (Parallel composition of systems)
Let c_1 and c_2 be two CSP-OZ-DC specifications, that agree only on the public channels from the interface part, which can be achieved by renaming local variables, declarations, and channels. Formally, if $State_i, PrivateInterface_i$, and $Paragraphs_i$ are the state schema, the private interface (i.e., the restriction of the interface to local channel declarations), and the paragraphs part of c_i for $i \in \{1, 2\}$, we demand

$$
\begin{aligned}
&dom\, State_1 \cap dom\, State_2 = \varnothing \\
&dom\, PrivateInterface_1 \cap dom\, PrivateInterface_2 = \varnothing \\
&\forall \mathcal{M}, \mathcal{M}_1, \mathcal{M}_2 : Model \mid (\mathcal{M}, \mathcal{M}_1) \in [\![Paragraphs_1]\!]^{\mathcal{D}}, \\
&\quad (\mathcal{M}, \mathcal{M}_2) \in [\![Paragraphs_2]\!]^{\mathcal{D}} \bullet \mathcal{M}_1 \cup \mathcal{M}_2 \in Model.
\end{aligned}
$$

We then define the parallel composition of CSP-OZ-DC systems *like in [Hoe06]:*

$$
\begin{aligned}
[\![c_1 \wedge c_2]\!]^{\mathcal{E}} \mathcal{M} \;:=\; &\{\mathcal{I}_1 : [\![c_1]\!]^{\mathcal{E}} \mathcal{M};\; \mathcal{I}_2 : [\![c_2]\!]^{\mathcal{E}} \mathcal{M} \\
&\mid \forall\, t : Time \bullet \mathcal{I}_1(t) \cup \mathcal{I}_2(t) \in Model \\
&\bullet \lambda\, t : Time \bullet \mathcal{I}_1(t) \cup \mathcal{I}_2(t)\}
\end{aligned}
$$

So, the semantics of the composition of two CSP-OZ-DC specifications contains all interpretations on which both components agree.

Remark 2.2.3 (Value and reference semantics). For the sake of simplicity, we also follow the convention of Hoenicke and Fischer to use the value semantics for pure OZ classes (i.e., without CSP and DC part) to avoid mutual dependencies if several classes share the same reference to an object. See [Fis00, Hoe06] for a detailed comparison of reference and value semantics in CSP-OZ and CSP-OZ-DC.

2.2.3. Related Combined Approaches

We have chosen to apply our approach exemplary to case studies modelled with CSP-OZ-DC, because it has some nice properties like its strict separation of concerns that allow modelling of the three system dimensions, control, data, and time, independently. In addition, CSP-OZ-DC has a compositional operational semantics that is presented in the next section. Simultaneously with the development of CSP-OZ and CSP-OZ-DC, there has been a lot of work in the integrated formal methods community on similar combined approaches: Real-time Object-Z [SH99, SH02] integrates

real-time constraints into OZ, and TCOZ [MD98] and RT-Z [Süh99] combine OZ and Z with Timed CSP [RR86, Sch95, Sch99], a CSP dialect with a time-out operator.

The untimed language CSP-OZ developed by Fischer [Fis97] is the basis for CSP-OZ-DC. Another combination of CSP with Z (without time) is Circus [WC01], which has been developed with a focus to step-wise refinement of specifications. Analogously, there are several combinations of Event-B [AM98] with CSP [But00, TS00, STW10] (also without time).

For more details on combined approaches related to CSP-OZ-DC and CSP-OZ, we refer to the work of Hoenicke or Fischer.

2.3. Phase Event Automata

2.3.1. Syntax and Semantics of Phase Event Automata

Phase Event Automata (PEA) are a class of timed automata [AD94] that describe the behaviour of state- and event-based systems. They are developed in [HM05b, Hoe06] as a model for the operational semantics of CSP-OZ-DC specifications.

In contrast to standard timed automata introduced by Alur and Dill [AD94], which only comprise event occurrences and timing constraints over real-valued variables called clocks, PEA also allow to specify properties over the data of a system. A distinguished feature of PEA is that parallel components synchronise on both data and events, which induces that the parallel composition of PEA is compositional with respect to safety properties. Due to this compositionality and the inclusion of events, data, and time, PEA are well-suited as a model for the operational semantics of CSP-OZ-DC.

Syntax of PEA

PEA inherit the notion of *clocks* from timed automata. A clock is a special variable that is automatically incremented as time passes. The following definition of PEA is based on [Hoe06], but differs in that we also allow symbolic constants (called *timing parameters*) to occur in clock invariants.

Definition 2.3.1 (Clock constraints)
For a set Clock of clock variables and timing parameters TimePar, the set $\mathcal{L}(C)$ of clock constraints with constants *is defined by the following BNF grammar:*

$$\theta ::= c < q \mid c \leq q \mid c < T \mid c \leq T \mid \theta \wedge \theta \mid \neg\theta,$$

where $c \in Clock$ is a clock, $q \in \mathbb{Q}^+$ is a rational constant, and $T \in TimePar$ is a timing parameter.

The set of convex *clock constraints $\mathcal{L}^c(C)$ is given by the following grammar:*

$$\theta^c ::= c < q \mid c \leq q \mid c < T \mid c \leq T \mid \theta^c \wedge \theta^c$$

The operator $strict(\theta)$ replaces all occurrences of $\leq$ in θ by $<$.

The semantics is given by clock valuations $\eta : \mathit{Clock} \to \mathit{Time}$, assigning values from a time domain to clocks. The set of all clock valuations is denoted by *ClVal*. Here, the time values are usually given by non-negative reals $\mathbb{R}^+$. The semantics of a timing parameter T is given by a model $\mathcal{M} : \mathit{TimePar} \to \mathbb{Q}^+$ assigning a rational constant to T that has an arbitrary value but is fixed over time. We write $\eta, \mathcal{M} \models \theta$ iff θ holds for η and $\mathcal{M}$. For a set of clocks X, we denote by $(\eta + t)$ the increasing of clocks, i.e., $(\eta + t)(c) := \eta(c) + t$, and by $\eta[X := 0]$ the valuation, where each clock in X is set to zero, and the values of the remaining clocks are given by η.

Let $\mathcal{L}(V)$ denote the set of predicates over variables in V. Its semantics is defined by models (or valuations) from *Model*, i.e., mappings $\mathcal{M} : \mathcal{L}(V) \to \mathcal{D}$ into a semantical domain $\mathcal{D}$.

Definition 2.3.2 (Phase Event Automaton)
A phase event automaton *is a tuple* $\mathcal{A} = (P, V, A, C, E, s, I, P^0)$*, where*

- P *is a finite set of* locations *with* initial locations $P^0 \subseteq P$*,*
- V *is a finite set of* system variables*,*
- A *is a set of Boolean* event variables*,*
- C *is a finite sets of real-valued* clocks*,*
- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathbb{P}(C) \times P$ *is a set of* transitions*,*
- $s : P \to \mathcal{L}(V)$ *associates with each location a* state invariant*,*
- $I : P \to \mathcal{L}^c(C)$ *associates with each location a convex* clock invariant*.*

A tuple $(p_1, \varphi, X, p_2) \in E$ represents a transition from p_1 to p_2 with a *guard* φ over (possibly primed) variables, clocks, and events, and with a set X of clocks that are reset when taking the transition. Primed variables v' denote the post-state of v, whereas v always refers to the pre-state. As predicates over clocks, the clock constraints from Def. 2.3.1 are allowed. In addition, we postulate the presence of a *stuttering transition* for every location of the automaton:

Definition 2.3.3 (Stuttering transition)
A transition $(p, \varphi, \varnothing, p)$ *is called* stuttering transition *if*

$$\bigwedge_{e \in A} \neg e \wedge \bigwedge_{v \in V} v' = v \wedge s(p) \wedge \mathit{strict}(I(p)) \Rightarrow \varphi.$$

This allows the automaton to take a transition doing nothing, i.e., no event occurs, no clock is reset, and no state variable is changed. This is particularly useful when defining the parallel composition of PEA below, because we do not have to distinguish between transitions that are executed synchronously and transitions that are executed by only one parallel component. Instead every transition step of a parallel composition

is executed synchronously, and every component that is not involved just executes a stuttering transition.

Note that in contrast to [Hoe06], our PEA definition does not incorporate initial transitions to define initial constraints on variables. Instead the definition presented here is in line with [MFR06, FJSS07, MFHR08] and uses initial locations like in finite automata or in standard timed automata [AD94, OD08].

Semantics of PEA

The operational semantics of PEA is given by *runs* over configurations of the shape

$$(p, Y, \mathcal{M}, \eta, t) \in P \times \mathbb{P}A \times \mathit{Model} \times \mathit{ClVal} \times \mathit{Time}.$$

Definition 2.3.4 (Run of a PEA)
A run *of a PEA $\mathcal{A}$ is a finite sequence of configurations*

$$\langle (p_0, E_0, \mathcal{M}_0, \eta_0, t_0), (p_1, E_1, \mathcal{M}_1, \eta_1, t_1), \ldots (p_n, E_n, \mathcal{M}_n, \eta_n, t_n) \rangle,$$

with locations $p_i \in P$, event sets $E_i \subseteq A$, models of variables $\mathcal{M}_i$, clock valuations η_i, and points in time $t_i \in \mathit{Time}$. Additionally, we demand that the following conditions are true:

- $p_0 \in P^0$
- $\mathcal{M}_i(a) \neq \mathcal{M}_{i+1}(a)$ *for all* $i \in 0..n-1$ *and* $a \in E_i$
- $\mathcal{M}_i \models s(p_i)$
- $\eta_0(c) = 0$ *for* $c \in \mathit{Clock}$
- $\eta_i + t_i \models I(p_i)$ *for all* $i \in 0..n$
- $t_i > 0$ *for all* $i \in 0..n$
- *for all* $i \in 0..n-1$ *there is a transition* $(p_i, \varphi, X, p_{i+1})$ *with*

  $$\mathcal{M}_i \cup \mathcal{M}'_{i+1}, \eta_i + t_i \models \varphi$$

  and $\eta_{i+1} = (\eta_i + t_i)[X := 0]$.

The intuition is that the run of the automaton starts in the location p_0. The automaton stays in the location p_i for t_i time units. During this time, the variables in V must not change their values, which are given by the model $\mathcal{M}_i$ and that satisfy the state invariant $s(p_i)$. The values of the clocks, when p_i is entered, are given by η_i. They are incremented by t_i when p_i is left. At that time the clock invariant $I(p_i)$ holds. The events in E_i occur and the guard φ holds when the transition from p_i to p_{i+1} is taken. Its unprimed variables are evaluated by the model $\mathcal{M}_i$ and the primed variables by $\mathcal{M}'_{i+1}$. The new value of the clock valuation, γ_{i+1}, is computed by adding the delay t_i for the current transition to the old clock valuation under consideration of the clock resets X of the transition.

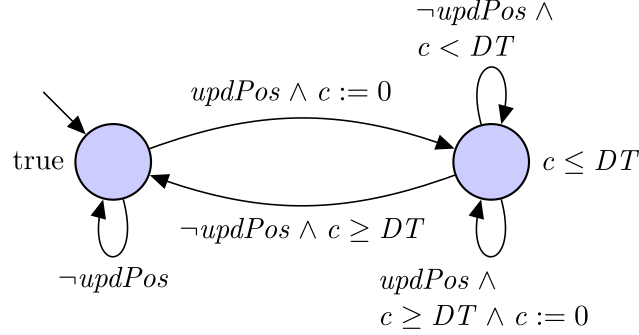


Figure 2.2.: Exemplary Phase Event Automaton

Example 2.3.5. Figure 2.2 pictures a PEA accepting the same interpretations as the DC formula

$$\neg\Diamond(\updownarrow \mathit{updPos}^\frown(\lceil \text{true} \rceil \wedge \ell < DT)^\frown \updownarrow \mathit{updPos}) \quad (2.3)$$

from the train example in Fig. 2.1. The left location is the initial location with invariant true. The automaton may stay in this location and execute arbitrary events except for *updPos*. An *updPos* event forces the automaton to enter the right location, by which the clock variable c is reset. Now, for DT time units the location cannot be left, and no *updPos* event can occur before the clock c reaches DT. The invariant of the location enforces that the location is left, or with another *updPos* event the clock is reset again.

Interpretations corresponding to runs. Since PEA are introduced to serve as operational semantics for CSP-OZ-DC, and PEA are also used in Chapter 6 to analyse timing properties of Verification Architectures, it is useful to examine which interpretations correspond to runs of PEA. We recall that an interpretation is a mapping assigning points in time to models of symbols, $\mathcal{I} : \mathit{Time} \to \mathit{Model}$.

Definition 2.3.6 (Run matches interpretation)
A run $\langle (p_0, E_0, \mathcal{M}_0, \eta_0, t_0^\Delta), \ldots (p_n, E_n, \mathcal{M}_n, \eta_n, t_n^\Delta)\rangle$ *matches an interpretation* $\mathcal{I}$ *iff for* $t_i = \sum_{k \in 0..i} t_k^\Delta$ *holds*

- $\mathcal{I}(t) = \mathcal{M}_0$ *for* $t < t_0$
- $\mathcal{I}(t) = \mathcal{M}_{i+1}$ *for* $i \geq 0$ *and* $t_i \leq t < t_{i+1}$
- $\mathcal{M}_i(a) \neq \mathcal{M}_{i+1}(a)$ *for all* $a \in E_i$ *and* $i \in 0..n-1$.

The sequence of models $\langle \mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$ is called untimed sequence of $\mathcal{I}$ and denoted by *untime*$(\mathcal{I})$. We write $\mathcal{I} \models \mathcal{A}$ iff $\mathcal{A}$ has a run that matches $\mathcal{I}$. So, we can

also relate PEA and DC formulae in the canonical way: a PEA *implements a DC formula,* $\mathcal{A} \models F$, iff they accept the same interpretations, i.e., for all $\mathcal{I}$

$$\mathcal{I} \models \mathcal{A} \Leftrightarrow \mathcal{I} \models F.$$

Parallel composition. PEA composed in parallel synchronise on common events and additionally on common variables. That is, a variable that occurs in both automata may only be changed if both automata agree. Clocks are not shared between the automata. The automata can always step synchronously, because any automaton can always take its stuttering transitions.

Definition 2.3.7 (Parallel composition of PEA)
The parallel composition *of PEA* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *with* $\mathcal{A}_l = (P_l, V_l, A_l, C_l, E_l, s_l, L_l, P_l^0)$ *with disjoint clock sets,* $C_1 \cap C_2 = \varnothing$*, is given by*

$$\mathcal{A}_1 \,||\, \mathcal{A}_2 := (P_1 \times P_2, V_1 \cup V_2, A_1 \cup A_2, C_1 \cup C_2, E, s_1 \wedge s_2, I_1 \wedge I_2, P_1^0 \times P_2^0),$$

where $((p_1, p_2), \varphi_1 \wedge \varphi_2, X_1 \cup X_2, (p_1', p_2')) \in E$ *iff* $(p_l, \varphi_l, X_l, p_l') \in E_l$ *for* $l = 1, 2$.

Hoenicke [Hoe06] shows that the parallel composition of two automata that are *stutter invariant*, i.e., where every location has a stuttering transition, is again stutter invariant.

Moreover, in [Hoe06] a very important compositionality result for PEA is proven, which we cite here.

Theorem 2.3.8 (Compositionality of PEA)
Given PEA $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *and DC formulae* F_1 *and* F_2 *with* $\mathcal{A}_1 \models F_1$ *and* $\mathcal{A}_2 \models F_2$ *then*

$$\mathcal{A}_1 \parallel \mathcal{A}_2 \models F_1 \wedge F_2.$$

This particularly means that any property that has been proven for one component is also valid for every parallel composition containing this component.

Sequential composition. In later chapters of this thesis, we also need a notion of sequential composition for PEA that correspond to sequential composition in CSP processes. To this end, we consider extended PEA with a final location:

$$\mathcal{A} = (P, V, A, C, E, s, I, P^0, f),$$

in which $f \in P$ is a single distinguished location marked as final. One could now define a specific acceptance condition over final locations, but for our purpose the final locations are only necessary as glue point for the sequential composition. The following definition is in line with, e.g., the definition from [OS10].

Definition 2.3.9 (Sequential composition of PEA)
The sequential composition *of PEA $\mathcal{A}_1$ and $\mathcal{A}_2$ with final locations, where $\mathcal{A}_l = (P_l, V_l, A_l, C_l, E_l, s_l, L_l, P_l^0, f_l)$, $l \in 1..2$, with disjoint locations and clock sets, $P_1 \cap P_2 = \varnothing$ and $C_1 \cap C_2 = \varnothing$, is given by*

$$\mathcal{A}_1 \fatsemi \mathcal{A}_2 := (P_1 \cup P_2 \cup P^g, V_1 \cup V_2, A_1 \cup A_2, C_1 \cup C_2, E, s_1 \cup s_2, I_1 \cup I_2, P_1^0, f_2).$$

The set of locations P^g is a set of new locations that is used to glue together both automata. It is defined by $P^g := \{f_1\} \times P_2^0$ with $s(f_1, p) = s(f_1) \wedge s(p)$ and $I(f_1, p) = I(f_1) \wedge I(p)$ for $(f_1, p) \in P^g$. The set of transitions E is defined by

$$\begin{aligned} E := {} & E_1 \setminus \{e \mid e = (p, \varphi, X, f_1) \in E_1\} \cup \\ & E_2 \setminus \{e \mid e = (p^0, \varphi, X, p) \in E_2, p^0 \in P_2^0\} \cup \\ & \{(p, \varphi, X \cup C_2, p^g) \mid (p, \varphi, X, f_1) \in E_1, p^g \in P^g\} \cup \\ & \{((f_1, p^0), \varphi, X, p) \mid (p^0, \varphi, X, p) \in E_2, p^0 \in P_2^0\}. \end{aligned}$$

2.3.2. Operational CSP-OZ-DC Semantics in Terms of PEA

In order to apply automated verification techniques, e.g., model checking [CGP99], to a CSP-OZ-DC specification, Hoenicke defines a translation from CSP-OZ-DC into PEA. The translation is compositional in the sense that every part of the CSP-OZ-DC specification is translated on its own:

$$PEA(\mathit{CSP\text{-}OZ\text{-}DC}) = PEA(\mathit{CSP}) \parallel PEA(\mathit{OZ}) \parallel PEA(\mathit{DC})$$

if *CSP-OZ-DC* is a CSP-OZ-DC specification with the corresponding parts *CSP*, *OZ*, *DC*. The DC part generally consists of several DC formulae $DC_1, \ldots, DC_n$ that are each translated into a single automaton, i.e.,

$$PEA(DC) = PEA(DC_1) \parallel \ldots \parallel PEA(DC_n)$$

CSP part. The translations of the CSP and the OZ part are simple. The CSP process is translated into a labelled transition system (see Sect. 2.1.1) and rewritten as PEA without data constraints and time. The LTS has to be finite to be verifiable by model checking. An example is given in Fig. 2.3.

OZ part. The PEA $PEA(OZ)$ belonging to an OZ specification OZ is given by $PEA(OZ) := (P, V, A, C, E, s, I, P_0)$, where

$P = \{p_0, p_1\}$

$V = \mathit{State}(var) \cup \{c : A \bullet c\bowtie\}$

$A =$ Set of channels with operation schemas.

$C = \varnothing$

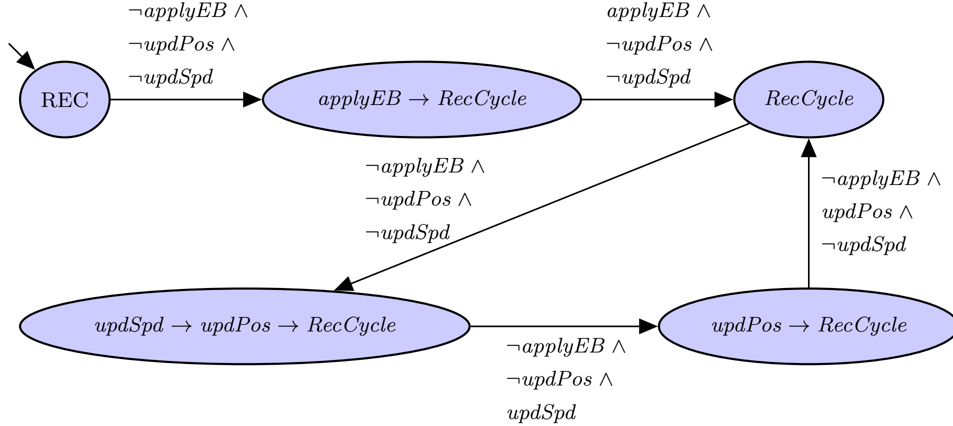


Figure 2.3.: A PEA representing the *REC* process of Fig. 2.1 (for the alphabet {applyEB, updPos, updSpd}; all invariants of the locations are true)

$$
\begin{aligned}
E = \{ & c : A \mid \\
& \quad c\bowtie = [q_1 : Q_1; \ \ldots ; \ q_n : Q_n] \wedge com_c = [\hat{q}_1 : Q_1; \ \ldots ; \ \hat{q}_n : Q_n \mid P] \wedge \\
& \quad (\forall\, i : 1..n \bullet \hat{q}_i \in \{q_i, q_i?, q_i!\}) \bullet \\
& \quad (p_1, only(c) \wedge (\textbf{let } \hat{q}_1 == c\bowtie'.q_1, \ldots, \hat{q}_n == c\bowtie'.q_n \bullet \texttt{com_}c), \varnothing, p_1)\} \\
& \cup \{i : \{1, 2\} \bullet (p_i, only(\tau) \wedge \Xi State, \varnothing, p_i)\} \\
& \cup \{(p_0, only(\tau) \wedge \Xi State, \varnothing, p_1)\} \\
s = \{ & p_0 \mapsto \texttt{Init}, p_1 \mapsto State\} \\
I = {} & \text{true} \\
P_0 = {} & \{p_0\}
\end{aligned}
$$

The Z term $\Xi State$ expresses that no symbol from the state schema is changed, i.e., an expression $v' = v$ is added for every variable v. The OZ part is translated into an automaton with two locations, one initial location, having the `Init` schema of the CSP-OZ-DC specification as invariant, and a second location with a loop transition for every operation of the CSP-OZ-DC class. The constraint of the state schema of the CSP-OZ-DC class becomes the invariant of both of the locations. So, the constraints of the operation schema are directly passed to the transitions and invariants of the PEA.

In Fig. 2.4, we present a simple example demonstrating the translation of the OZ part for a CSP-OZ-DC specification.

DC part. The most difficult part is the translation of the DC formulae. Hoenicke [Hoe06] defined a translation from DC counterexample traces into PEA. When trans-

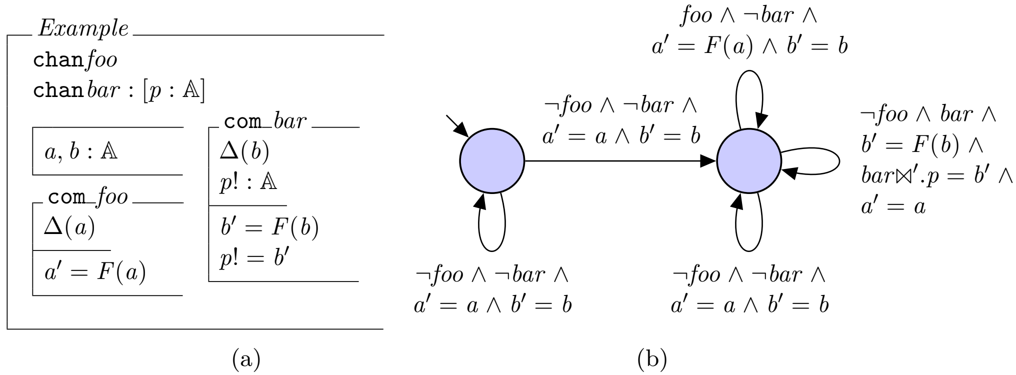


Figure 2.4.: Translation of the OZ part (a) into PEA (b)

lating such a formula into an automaton, the resulting automaton has to decide whether a run matches all of the consecutive phases of a trace formula. The problem arises that the consecutive phases are not mutually exclusive in the sense that at any point in time several phases are potentially active. For instance, in formula (2.3) from Example 2.3.5 when the automaton detects the first *updPos* event, then the second trace ($\ell < DT$) can potentially be active or the first true phase can be active. So, the automaton has to reflect all possible combinations of phases by which a power-set construction becomes necessary, similarly to the construction of a deterministic automaton from a non-deterministic automaton. By this, the size of the resulting automaton is exponential in the length of the DC counterexample traces. [MFHR08] gives an overview over this construction. The automaton of Fig. 2.2 is exactly the outcome of the automated PEA construction of Hoenicke when applied to formula (2.3).

Note that the translation of Hoenicke can directly be carried over to DC formulae and PEA with symbolic clock constants instead of concrete rational constants [FJSS07]: Since the translation of Hoenicke and, analogously, the soundness and compositionality results do not depend on having concrete values for rational constants, we can treat timing parameters exactly like rational constants in the translation.

Compositionality of CSP-OZ-DC

Hoenicke showed in [Hoe06] that the compositional PEA semantics of CSP-OZ-DC is sound, i.e., every part of the specification can be translated on its own into a PEA, and the parallel composition is computed afterwards. Thus, the compositionality result for PEA from Thm. 2.3.8 holds similarly for CSP-OZ-DC specifications: if a property is proven correct for one part of the specification, then it is also correct for the entire CSP-OZ-DC specification. This enables modular reasoning, because often it suffices to consider, e.g., only the DC part to prove timing properties.

3 Extended CSP for Verification Architectures

> We do what we must, Lucien. Sometimes we can choose the path we follow. Sometimes our choices are made for us. And sometimes we have no choice at all.
>
> *(Dream, in* Season of Mists, *Neil Gaiman)*

3.1. CSP Processes for Verification Architectures 42
3.1.1. CSP Processes with Data Constraints 43
3.1.2. Unknown Processes 47
3.1.3. Assumptions on Unknown Processes 49
3.1.4. Running Example: A Train Control Protocol 50
3.2. Properties of Extended CSP 52
3.2.1. Continuous CSP Operators 52
3.2.2. Discontinuity of Parallel Composition 54
3.3. Normal Forms 56
3.4. Discussion 60
3.4.1. Parametric Systems 60
3.4.2. Semantics 61
3.4.3. Related work 62

This chapter introduces the CSP dialect that we use to describe VAs.

3.1. CSP Processes for Verification Architectures

We firstly define an extension of CSP by data types; then, in a further extension we introduce constrained unknown processes that are useful to model Verification Architectures. We denote this extension of CSP by data constraints and unknowns *eCSP*.

Terms and formulae. For specifying constraints over data types within CSP processes and our dynamic logic extension dCSP, we consider many-sorted first-order formulae with predicates and function symbols from a signature

$$\Sigma = (Sort, SysVar, Const, Var),$$

where *Sort* is a set of sorts, *SysVar* is a set of (primed and unprimed) system variables and function symbols with sorts from *Sort*. *Const* is a set of unprimed constants or parameter symbols, and *Var* is a set of variables. In contrast to the system variables, the variables in *Var* do not describe the global state of the system but instead are used as local variables in constraints, e.g., they are applied to define parameters of events. Predicate symbols are modelled as special function symbols of sort $\mathbb{B}$. We distinguish between non-rigid functions *SysVar* that can be changed over time and constants (or parameters) *Const* that are time-independent. We denote terms over this signature with $Term_\Sigma$ and formulae with $Form_\Sigma$. In the following, we demand that every signature Σ contains the function and predicate symbols $+, -, *, /, =, <, >, \leq, \geq$ for arithmetical operations with the usual fixed interpretations. We write $SysVar(P)$ or $SysVar(\gamma)$ to denote the set of all system variables in a process P or a formula γ, respectively.

Interpretation of function symbols. We express updates of system variables using the primed notation: in an expression

$$x' = x + c$$

the primed variant of the system variable x refers to the post-state of x and the unprimed variant refers to the pre-state of x. We make a commitment that *SysVar* implicitly contains a primed symbol for every unprimed symbol whereas *Const* and *Var* do not contain any primed symbols. Note that with respect to the signature Σ, the system variable x is a non-rigid function symbol with arity 0 from *SysVar*. This means that the valuation of the system variable x can change over time, which is modelled by sequences of valuations for system variables. On the other hand, the constant symbol c is always interpreted in the same way during a system run. If such a constant symbol is not fixed to a specific value we denote it as *parameter* of a system. Moreover, standard symbols like $+$ are function symbols from *Const* with arity 2. As a convention, these symbols are interpreted in the same and usual way for all possible system runs. One advantage of this proceeding is that we can model data structures like arrays and lists as functions from *SysVar* that may change over time.

Semantics of terms and formulae. The semantical domain for a signature $\Sigma = (Sort, SysVar, Var, Const)$ is given by sets of values $\mathcal{D}_S$ for sorts $S \in Sort$. A model (or valuation) for variables, system variables, or parameters of sort S is a (partial) mapping $\mathcal{M} : Var \nrightarrow \mathcal{D}_S$, $\mathcal{M} : SysVar \nrightarrow \mathcal{D}_S$, or $\mathcal{M} : Const \nrightarrow \mathcal{D}_S$, respectively. A model $\mathcal{M}$ of an n-ary function symbol $f : S_1 \times \cdots \times S_n \to S$ is a function on the data domain: $\mathcal{M}(f) : \mathcal{D}_{S_1} \times \cdots \times \mathcal{D}_{S_n} \to \mathcal{D}_S$. We consider only models that interpret standard symbols like $+, -, *, /, =, <, >, \leq, \geq$ etc. in the usual way and denote the set of all such models by *Model*. The semantics of a term $f(\theta_1, \ldots, \theta_n)$ is a mapping $[\![\cdot]\!] : Model \to \mathcal{D}_S$ with

$$[\![f(\theta_1, \ldots, \theta_n)]\!]\mathcal{M} = \mathcal{M}(f)([\![\theta_1]\!]\mathcal{M}, \ldots, [\![\theta_n]\!]\mathcal{M}).$$

The semantics of atomic formulae is defined analogously:

$$\mathcal{M} \models p(\theta_1, \ldots, \theta_n) \quad \text{iff} \quad \mathcal{M}(p)([\![\theta_1]\!]\mathcal{M}, \ldots, [\![\theta_n]\!]\mathcal{M})$$

The semantics of quantifiers and Boolean connectives are defined as usual. We use the convention that $\mathcal{M}$ is a model only for unprimed function symbols, while $\mathcal{M}'$ is a model for primed symbols.

With $\mathcal{M}[v := d]$ we denote the *substitution* of a symbol v in the model $\mathcal{M}$. The model $\mathcal{M}[v := d]$ does agree with $\mathcal{M}$ except for v that is evaluated to d:

$$\mathcal{M}[v := d](u) := \begin{cases} d & \text{if } u = v \\ \mathcal{M}(u) & \text{otherwise.} \end{cases}$$

3.1.1. CSP Processes with Data Constraints

As in standard CSP [Hoa85], we consider processes synchronising on *structured* events, i.e., events are combinations of a channel with some messages as parameter: $c.v_1.v_2$ behaves like a simple event that can be used for synchronisation. It consists of the channel c and some values or messages v_1, v_2. Here, channels have a unique sort. With a channel declaration $c : S_1 \times \cdots \times S_n$ the channel c can be used to communicate values of sort S_1 up to S_n, e.g., $c.v_1.\ldots.v_n$ is an event of this channel if $v_i : S_i$ for $i \in 1..n$. We denote the sort of a channel c by $sort(c)$.

As usual, we define abbreviations $c!y$ to express the sending of a value in variable $y!$ and $c?x$ to express the receiving of a value into the variable $x?$. We use the convention from CSP-OZ-DC that output variables are always indicated by ! and input variables by ?. We call $x!$ and $y?$ *message variables of* c. If the channel is declared as a channel of sort $c : S$, then x has to be a variable of sort S. We denote the set of all channels by *Channels*. The set of all events is then defined by[1]

$$Events == \{c : Channels;\ v_1 : S_1;\ \ldots;\ v_n : S_n \mid sort(c) = S_1 \times \cdots \times S_n \bullet c.v_1.\ldots.v_n\}.$$

[1] Using standard Z syntax for set expressions (similarly to the schema definitions in Sect. 2.1.2), see [Spi92, ISO02] for details.

Like in [Hoe06], we use special parameter variables $c\bowtie_i : S_i$ with $i \in 1..n$ that hold the values of the messages of a channel $c : S_1 \times \cdots \times S_n$. The distinguished symbol $\bowtie$ always indicates parameter variables and shall not be used elsewhere.

The alphabet of a process P, i.e., the set of events on which P may communicate, is given by $alph(P)$.

Syntax of CSP Processes with Data Constraints

The syntax of *CSP processes with data* over a set of events *Events* is then given by the following BNF:

$$P ::= \texttt{Stop} \mid \texttt{Skip} \mid (a \bullet \varphi) \to P \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid P_1 ||| P_2 \\ \mid P_1 \parallel_A P_2 \mid P_1 \,{}_{9}^{\circ}\, P_2 \mid X,$$

where $a \in Events$, $A \subseteq Events$ and φ is a formula from $Form_\Sigma$.

In this definition, the difference to the standard CSP definition is that we have constrained occurrences of events $a \bullet \varphi$. The intuition is that when the event a occurs, the state space is changed according to the constraint φ, where unprimed function symbols in φ refer to the valuations before the occurrence of a and primed function symbols to the valuations after a. If a is a structured event with messages, the constraint φ may also include these variables, e.g.,

$$a!x?y \bullet x! = y? + 1$$

is a valid constrained event with messages. As a notational convention, we write φ' to denote a constraint that is equal to φ except that every unprimed function symbol x is replaced by x'. If not stated otherwise, we write a instead of $a \bullet \text{true}$. As usual, a process may also be a call of a process identifier X. In this case, there has to be a unique process declaration $X \stackrel{c}{=} F$, which can also be a recursive definition $X \stackrel{c}{=} F(X)$, where $F(X)$ is a guarded process containing references to X. As in [Hoa85], we denote the solution of this process equation by $\mu\, X \bullet F(X)$ and demand that $F(X)$ is a guarded process, i.e., a process that begins with an event (thus, $F(X)$ does not engage in infinite internal steps). For simplicity, we restrict ourselves to processes that can be represented without hiding and renaming in order to avoid infinite non-determinism that can be introduced with these operators.

Additionally, we use *replicated* CSP operators [Fis00, Hoe06] as an abbreviation: for a set of events E the following BNF describes processes with replicated operators.

$$P ::= (\Box\, e : E \bullet P) \mid (\big\|_A\, e : E \bullet P) \mid (\big|\big|\big|\, e : E \bullet P) \mid ({}_{9}^{\circ}\, e : E \bullet P)$$

We demand that the set E does not introduce non-deterministic infinite branching. That is, for all processes P and each event a the set

$$\{Q \mid P \stackrel{a}{\longrightarrow} Q\}$$

has to be finite. Therefore, choices like $\Box\, n : \mathbb{N} \bullet (a \to P_n)$ are not allowed.

Sometimes it is convenient to throw away the constraints within a CSP process to get a standard CSP process. Hence, we define a function *unconst* returning the unconstrained CSP process for a given CSP process with data constraints:

$$unconst(P) := \begin{cases} a \to unconst(Q) & \text{if } P = (a \bullet \varphi) \to Q \\ unconst(P_1)\ op\ unconst(P_2) & \text{if } P = P_1\ op\ P_2 \\ P & \text{otherwise,} \end{cases}$$

where $op \in \{\Box, \|, |||, \sqcap, ;\}$.

Remark 3.1.1. CSP processes may generally be constructed by infinite process terms. But we restrict ourselves, here and in what follows, to processes with an inductively constructed syntax. Thus, we only consider replicated process terms like defined above to obtain inductive process terms, in opposite to [Ros98], where non-deterministic choices over arbitrary process sets are possible. We already excluded non-deterministically infinite-branching processes. With these restrictions to an inductively defined syntax, we are able to use structural induction over process terms in our proofs.

Semantics of CSP Processes with Data Constraints

The semantics of CSP processes with data constraints is given here by interpretations $\mathcal{I}$, mappings from a time domain *Time* (usually $\mathbb{N}$ or $\mathbb{R}$) into the set of all models: $\mathcal{I} : Time \to Model$. A model (or valuation) $\mathcal{M}$ for system variables and parameters of sort S is a (partial) mapping into a corresponding domain:

$$\mathcal{M} : (SysVar \cup Const) \nrightarrow \mathcal{D}_S.$$

We denote the set of all interpretations by *Interpretation*. So, an interpretation is a timed sequence of models that corresponds to state changes performed by constrained event occurrences. Every interpretation belongs to a run of the CSP process. Events are modelled as Boolean variables that change their values if a corresponding event occurs (cf. Sect. 2.1.3, page 25). To compute the semantics, we first compute the labelled transition system (LTS) of the CSP process in the standard way as if there were no data part (see Sect. 2.1.1) but with events that are annotated with data constraints. That is, the events of the LTS are compound events of the shape $a \bullet \varphi$, which are treated as standard LTS events. Only for the parallel composition a different LTS rule is used that conjoins the constraints of synchronising events, but apart from that, it is identical to the standard rule:

$$\frac{P \xrightarrow{a\bullet\varphi} P' \quad Q \xrightarrow{a\bullet\psi} Q'}{P \underset{A}{\|} Q \xrightarrow{a\bullet\varphi\wedge\psi} P' \underset{A}{\|} Q'} \qquad a \in A \setminus \{\tau\} \qquad \text{(constrained parallel)}$$

We say that an interpretation $\mathcal{I}$ *fits* to a run $\pi = \langle (a_1 \bullet \varphi_1), (a_2 \bullet \varphi_2), \dots \rangle$ of the LTS iff there are a points $t_0, t_1, \dots \in$ *Time* with $0 = t_0 < t_1 < t_2 < \dots$ and models $\mathcal{I}(t) = \mathcal{M}_i$ for $t_i \leq t < t_{i+1}$ such that

$$\mathcal{M}_i(a_{i+1}) \neq \mathcal{M}_{i+1}(a_{i+1}) \text{ for } i \geq 0,$$

where all a_i are of sort $\mathbb{B}$ in the signature Σ. We call the sequence $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$ *untimed sequence* of $\mathcal{I}$, denoted by $\mathit{untime}(\mathcal{I})$.

On this basis, we define the semantics of a process: $\mathcal{I} \in [\![P]\!]\mathcal{M}$ iff

1. there is a run $\pi = \langle a_1, a_2, \dots \rangle$ of the LTS of $\mathit{unconst}(P)$ such that $\mathcal{I}$ fits to π. Let the resulting untimed sequence be $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$.
2. $\mathcal{M}_0 = \mathcal{M}$
3. $(\mathcal{M}_{i-1} \cup \mathcal{M}'_i) \models \varphi_i$ for $i > 0$
4. $\mathcal{M}_i(v) = \mathcal{M}_{i+1}(v)$ for all parameter $v \in \mathit{Const}$ and $i \geq 0$
5. if $a_i = \checkmark$ then $\mathcal{M}_{i-1}(v) = \mathcal{M}_i(v)$ for all symbols $v \in \mathit{SysVar}$,

where $\mathcal{M}'_i$ is a model for primed symbols, i.e., $\mathcal{M}'_i(f') = \mathcal{M}_i(f)$.

Note that this definition makes use of the trace semantics and not the failures-divergences semantics of CSP. If not stated otherwise, we always have the trace semantics in mind. One consequence is that we do not have to distinguish external choice and internal choice that are semantically equal in trace semantics. But many results likewise hold for the stable failures model.

Notation. An interpretation $\mathcal{I}$ maps points in time to models. But often the exact points in time of event occurrences do not matter, only the order in which events occur and the models that are changed. Thus, for the sake of conciseness, when we write

$$\mathcal{I} = \langle \mathcal{M}_0, a_1, \mathcal{M}_1, a_2, \dots \rangle$$

we mean an interpretation with $\mathcal{I}(0) = \mathcal{M}_0$ and events $a_1, a_2, \dots$ that occur in the specified order at arbitrary points in time and that change the models according to the definition of the semantics above. To express the occurrence of an event between two consecutive models $\mathcal{M}$ and $\mathcal{N}$, we write $\mathcal{M} \cup \mathcal{N}' \models \updownarrow e$ iff $\mathcal{M}(e) \neq \mathcal{N}(e)$.

In order to avoid redundancies, we write $\langle \dots, \mathcal{M}, \checkmark \rangle$ instead of $\langle \dots, \mathcal{M}, \checkmark, \overline{\mathcal{M}} \rangle$ to denote that an interpretation represents a terminated run of a process. We omit the model after $\checkmark$, because it is equal to $\mathcal{M}$ except for the valuation of the $\checkmark$-event.

The concatenation of interpretations $\mathcal{I}_1 = \langle \dots, \mathcal{M} \rangle, \mathcal{I}_2 = \langle \overline{\mathcal{M}}, b, \dots \rangle$, where particularly $\mathcal{I}_1(t) = \mathcal{M}, \forall\, t \geq t_0$, is denoted by $\mathcal{I}_1 \frown \mathcal{I}_2$ with the meaning

$$(\mathcal{I}_1 \frown \mathcal{I}_2)(t) := \begin{cases} \mathcal{I}_1(t) & \text{if } t < t_0 \wedge \mathcal{M} = \overline{\mathcal{M}} \\ \mathcal{I}_2(t - t_0) & \text{if } t \geq t_0 \wedge \mathcal{M} = \overline{\mathcal{M}} \\ \text{undefined} & \text{else.} \end{cases}$$

The symbol $\equiv$ denotes equivalence—with respect to interpretations—of processes, also called *interpretation-equivalence*. That is, $P_1 \equiv P_2$ means that for all models $\mathcal{M}$ the sets of interpretations for P_1 and P_2 are equal: $[\![P_1]\!]\mathcal{M} = [\![P_2]\!]\mathcal{M}$. We use the symbol $=$ to denote syntactical equality on process terms. Sometimes we also need the classical *trace-equivalence* on processes not considering the data constraints. To this end, we use the symbol $P_1 \equiv_T P_2$ iff $\mathit{traces}(P_1) = \mathit{traces}(P_2)$, where the constrained events $a \bullet \varphi$ are treated as compound events.

3.1.2. Unknown Processes

To be able to specify Verification Architectures we need a higher degree of freedom than in the CSP definitions of Sect. 3.1.1. To this end, we additionally introduce *unknown processes* [Fab09, Fab10a, Fab10b]. Unknown processes are special processes that allow the occurrence of arbitrary events except for events from a fixed alphabet and arbitrary changes of variables except for variables from a fixed set. An unknown process can also choose to terminate. Additionally, these unknown processes may be restricted by constraints from an *arbitrary* logic (at least, if this logic has the same semantical domain as CSP with data constraints). On the level of CSP, these constraints are handled as black boxes that restrict the possible behaviour of a process. Those *constrained* unknown processes are introduced in the next section; we start by defining unconstrained unknown processes here.

Syntax of CSP with Unknown Processes

The syntax of *CSP with unknown processes*, denoted eCSP, is an extension of the syntax from Sect. 3.1.1 by

$$P ::= \mathtt{Proc}_{\setminus A,V} \mid \mathtt{Proc}^{\infty}_{\setminus A,V},$$

where $A \subseteq \mathit{Events}$ and $V \subseteq \mathit{SysVar}$.

An unknown process $\mathtt{Proc}_{\setminus A,V}$ can perform arbitrary events except for events from the given set A. In addition, all variables from set V are not allowed to be changed in the execution of the process. An unknown process marked with the ∞-symbol $\mathtt{Proc}^{\infty}$ will never terminate. We use $\mathtt{Proc}^{(\infty)}_{\setminus A,V}$ if we refer to either $\mathtt{Proc}_{\setminus A,V}$ or $\mathtt{Proc}^{\infty}_{\setminus A,V}$. An unknown process $\mathtt{Proc}_{\setminus A,V}$ can be rewritten to

$$\mathtt{Proc}_{\setminus A,V} = \mathtt{Skip} \,\Box\, \big(\,\Box\, a : \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A \bullet (a \bullet \Xi V \rightarrow \mathtt{Proc}_{\setminus A,V})\big),$$

where $\mathbb{U}^{\tau}_{\mathit{Events}}$ is the universe of all possible events including τ (in contrast to $\mathbb{U}_{\mathit{Events}}$, the universe of events without τ). Since the operational semantics of CSP processes is given by labelled transition systems, we need to extend the set of transition rules constituting the LTS semantics of a process.

Transition Rules for Unknown Processes

The set of transition rules for computing the LTS of a CSP process is extended by the following rules to cope with unknown processes:

$$\frac{}{\texttt{Proc}^{(\infty)}_{\setminus A,V} \xrightarrow{a \bullet \Xi V} \texttt{Proc}^{(\infty)}_{\setminus A,V}}, \qquad \text{where } a \in \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A$$

The process can perform an arbitrary event that is not in the set A, and the constraint of the event has to ensure that symbols from the set V are not changed, which is expressed in Z syntax by ΞV.

$$\frac{}{\texttt{Proc}_{\setminus A,V} \xrightarrow{\checkmark} \Omega}$$

If the process is not marked as infinite unknown process, it may non-deterministically decide to terminate.

We constitute our claim of the alternative representation of unknown processes given above in the following theorem.

Theorem 3.1.2 (Equivalent representation of unknown processes)
Unknown processes can equivalently be represented as

$$\texttt{Proc}_{\setminus A,V} \equiv \mu X \bullet \texttt{Skip} \,\Box\, (\,\Box\, a : \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A \bullet (a \bullet \Xi V \to X))$$

and

$$\texttt{Proc}^{\infty}_{\setminus A,V} \equiv \mu X \bullet \Box\, a : \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A \bullet (a \bullet \Xi V \to X).$$

Proof. We consider the process $P_1 = \texttt{Proc}_{\setminus A,V}$ as defined in Sect. 3.1.2 and

$$P_2 = \mu X \bullet \texttt{Skip} \,\Box\, (\,\Box\, a : \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A \bullet (a \bullet \Xi V \to X).$$

We prove the stronger result that P_1 and P_2 are equivalent even in stable-failures semantics[2]. For this purpose, we have to show that the traces and the failures sets of both of the processes coincide.

Let the corresponding labelled transition systems be $\mathit{LTS}(P_1)$ and $\mathit{LTS}(P_2)$. The initial state of $\mathit{LTS}(P_1)$ is $\texttt{Proc}_{\setminus A,V}$. There are the following possible transitions from $\texttt{Proc}_{\setminus A,V}$:

$$\texttt{Proc}_{\setminus A,V} \xrightarrow{a \bullet \Xi V} \texttt{Proc}_{\setminus A,V} \quad \text{and} \quad \texttt{Proc}_{\setminus A,V} \xrightarrow{\checkmark} \Omega \tag{3.1}$$

for all $a \in \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A$; there are no other locations than $\texttt{Proc}_{\setminus A,V}$. From this we get sets of traces

$$\begin{aligned} \mathit{traces}(P_1) = \{ & w : (\mathbb{U}_{\mathit{Events}} \setminus A)^* \bullet \langle w \rangle \} \cup \\ \{ & w : (\mathbb{U}_{\mathit{Events}} \setminus A)^* \bullet \langle w, \checkmark \rangle \}. \end{aligned}$$

[2] A *failure* in the stable-failures semantics of CSP is a tuple of a trace and a set of events that can be refused by the process after executing the trace.

Since the LTS can always perform a $\checkmark$-event in the only location $\texttt{Proc}_{\setminus A,V}$, it has no stable[3] location. Thus, to compute the failures of P_1, we only need to consider all finite traces that end up in states where a $\checkmark$ is possible, which are all finite traces. The $\checkmark$ causes that every set of events may be refused:

$$\mathit{failures}(P_1) = \{w : \mathit{traces}(P_1), X \subseteq \mathbb{U}^{\checkmark}_{\mathit{Events}} \bullet (\langle w, \checkmark \rangle, X)\}.$$

The process P_2 consists of an external choice. The single parts can perform the transition steps

$$\texttt{Skip} \xrightarrow{\checkmark} \Omega$$

$$(\,\Box\, a : \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A \bullet (a \bullet \Xi V \to X)) \xrightarrow{b \bullet \Xi V} X$$

for all $b \in \mathbb{U}^{\tau}_{\mathit{Events}} \setminus A$. So, the entire external choice, i.e., the process P_2, may perform the same steps: $P_2 \xrightarrow{\checkmark} \Omega$ and $P_2 \xrightarrow{b \bullet \Xi V} X$. The process reference X can be resolved by $X \xrightarrow{\tau} P_2$.

In contrast to $\mathit{LTS}(P_1)$, the LTS of P_2 has two locations, but again there are no stable locations, because it is always possible to execute a τ or a $\checkmark$-step. The traces of $\mathit{LTS}(P_2)$ are arbitrary finite traces of events from $\mathbb{U}_{\mathit{Events}} \setminus A$, and after every trace the process may terminate. Thus, $\mathit{traces}(P_2) = \mathit{traces}(P_1)$. Since both of the LTS have no stable locations and both of the processes have the same alphabet, we also get $\mathit{failures}(P_2) = \mathit{failures}(P_1)$, which finishes this proof. $\Box$

We could have used the alternative representation as definition for unknown processes. Anyway, we have introduced the semantics of unknown processes by means of its own transition rules for the operational semantics, because it makes the handling of unknown processes in proofs easier.

3.1.3. Assumptions on Unknown Processes

If we always allow nearly arbitrary behaviour for unknown CSP processes, it will be hard to prove anything useful over unknown CSP processes. Therefore, unknown processes will usually occur in a context where the behaviour of a system is constrained by additional temporal formulae. Hence, we introduce unknown CSP processes with additional constraints.

Syntax of constrained unknown processes. The syntax of unknown CSP processes with additional constraints is given by the following BNF.

$$P ::= (\texttt{Proc}_{\setminus A,V} \bullet F) \mid (\texttt{Proc}^{\infty}_{\setminus A,V} \bullet F),$$

where $A \subseteq \mathit{Events}$, $V \subseteq \mathit{SysVar}$ and F is a temporal formula. The only restriction we impose on the logic of F is that it has to be defined in the same semantical domain

[3] A location is *stable* if it has no outgoing τ or $\checkmark$-transitions.

as CSP with data constraints, i.e., its semantics has to be given by interpretations $[\![\cdot]\!] : \mathit{Model} \to \mathit{Interpretation}$.

The intuition behind an unknown process like $\mathtt{Proc}_{\backslash\{a,b\},\{v\}} \bullet F$, where F is, e.g., a DC formula, is that during the execution of the process arbitrary behaviour is allowed provided that the formula F is not violated. In addition, the events a and b are forbidden and the variable v cannot be changed in this execution. Like in the previous section the ∞-symbol indicates that the process will never terminate.

The semantics of these additionally constrained processes is given by the interpretations that are permitted by both the unknown process and the constraining formula.

Semantics of constrained unknown processes. The semantics of an constrained process is given by interpretations

$$\mathcal{I} \in [\![\mathtt{Proc}^{(\infty)}_{\backslash A, V} \bullet F]\!]\mathcal{M}$$

iff

- $\mathcal{I} \in [\![\mathtt{Proc}^{(\infty)}_{\backslash A, V}]\!]\mathcal{M}$ and
- $\mathcal{I}$ is in the semantics of F: $\mathcal{I} \in [\![F]\!]$.

The latter is well-defined because we have demanded that the semantical domain of F is compatible with the semantics of CSP with data constraints.

The semantics of a constrained unknown process in the context of a CSP expression can then be computed by exploiting that the trace semantics is a congruence for the corresponding operators, which is further examined in the next section. So, to compute the semantics of $P \boxtimes \mathtt{Proc}_{\backslash A, V} \bullet F$ with $\boxtimes \in \{\Box, \fatsemi\}$, we lift the operators to the interpretation level: if $\mathcal{I}_1 \in [\![P]\!]\mathcal{M}$ and $\mathcal{I}_2 \in [\![\mathtt{Proc}_{\backslash A, V} \bullet F]\!]\mathcal{M}$, then $(\mathcal{I}_1 \boxtimes \mathcal{I}_2) \in [\![P \boxtimes \mathtt{Proc}_{\backslash A, V} \bullet F]\!]\mathcal{M}$. Unfortunately, this does not work for parallel composition of processes, because the set of interpretations of a parallel composition cannot be computed from the interpretations of its constituents. Thus, we focus on processes without parallel compositions over unknowns at first and instead analyse parallel compositions over unknowns in Sect. 6.1.

For convenience, we will always write $\mathtt{Proc}^{(\infty)}_{\backslash A, V}$ instead of $\mathtt{Proc}^{(\infty)}_{\backslash A, V} \bullet \mathrm{true}$, and with $\mathtt{Proc}^{(\infty)}_{\backslash A, V} \bullet \varphi$ we implicitly refer to both unconstrained and constrained unknown processes.

3.1.4. Running Example: A Train Control Protocol

In the introduction (Sect. 1.1), a small train control system was presented, realising a simple movement authority (MA) procedure for a train and an RBC. The purpose of the protocol is to allow a train for requesting extensions of an MA while ensuring that the train never exceeds its current MA.

For the part of the architecture, we keep the safety of the system abstract: we use a real-valued variable *sf* containing a safety value for the system, that is filled with a concrete meaning when instantiating the system. For the purpose of defining a safe architecture it is sufficient to demand that *sf* is never lesser than 0.

The signature of the VA is given by

$$\Sigma = (\{\mathbb{R}, \mathbb{B}, \mathit{Time}\}, \{\mathit{sf} : \mathbb{R}, \mathit{ok} : \mathbb{B}\}, \{\mathit{RD} : \mathbb{R}, \mathit{CT} : \mathit{Time}\}, \varnothing),$$

using the sorts $\mathbb{R}$ and $\mathbb{B}$ and a time sort *Time*. Besides the safety value *sf*, there is only one other variable *ok*. In addition, two constants are declared: *RD*, which contains a safety distance to the end of the MA, and a time constant *CT*.

With $A = \{\mathit{check}, \mathit{fail}, \mathit{pass}, \mathit{extend}\}$ and $C = \{\mathit{RD}, \mathit{CT}\}$ the process defining the VA is given by

$$\begin{aligned}
\mathit{System} &\overset{c}{=} \big(\mathit{FAR} \fatsemi \mathit{check} \bullet \varphi_{\mathit{check}} \\
&\qquad \to (\mathit{fail} \bullet \varphi_{\mathit{fail}} \to \mathit{REC} \;\Box\; \mathit{pass} \bullet \varphi_{\mathit{pass}} \to \mathit{System})\big) \\
&\quad\; \Box \\
&\quad\; \big(\mathit{extend} \bullet \varphi_{\mathit{extend}} \to \mathit{System}\big) \\
\mathit{FAR} &\overset{c}{=} \mathtt{Proc}_{\backslash A,C} \bullet F_{\mathit{FAR}} \\
\mathit{REC} &\overset{c}{=} \mathtt{Proc}^{\infty}_{\backslash A,C} \bullet F_{\mathit{REC}}.
\end{aligned}$$

The *System* process consists of a choice over two sub-processes: one extends the MA by performing an *extend* event and the other executes a check cycle. The system phase *FAR*, modelled as unknown process, represents the situation where the train is at a safe distance to the end of the authority. Periodically, a *check* event is executed and based on its output the *System* process is recursively started again or the *REC* phase is entered, bringing the train into a safe recovery state (and it is never left again in this simplifying protocol). *REC* is defined as an unknown process that never terminates. Both unknown processes can change the system variables *sf* and *ok* arbitrarily, but the symbols *RD* and *CT* are kept constant. No event from *A* can occur in *FAR* or *REC*. The constraints of this process are given by

$$\begin{aligned}
\varphi_{\mathit{check}} &= \Xi(\mathit{sf}) \wedge \mathit{sf} \leq \mathit{RD} \wedge \neg \mathit{ok}' \vee \Xi(\mathit{sf}) \wedge \mathit{sf} > \mathit{RD} \wedge \mathit{ok}' \\
\varphi_{\mathit{fail}} &= \Xi(\mathit{sf}) \wedge \neg \mathit{ok} \\
\varphi_{\mathit{pass}} &= \Xi(\mathit{sf}) \wedge \mathit{ok} \\
\varphi_{\mathit{extend}} &= \mathit{sf}' > \mathit{sf} \\
F_{\mathit{FAR}} &= \neg \Diamond (\lceil \mathit{sf} > \mathit{RD} \rceil \frown \ell < \mathit{CT} \frown \lceil \mathit{sf} \leq 0 \rceil) \wedge \\
&\quad\; \neg \Diamond (\ell > \mathit{CT}) \\
F_{\mathit{REC}} &= \neg \Diamond (\lceil \mathit{sf} > 0 \rceil \frown \lceil \mathit{sf} \leq 0 \rceil).
\end{aligned}$$

The notation $\Xi(sf)$ is adapted from the Z language [Spi92], and it denotes a system variable that is not changed, $\Xi(sf) \equiv sf' = sf$. The constraint φ_{check} sets the new value of *ok*, i.e., *ok'*, depending on whether *sf* is greater or lesser than the safety distance *RD*. The movement authority is extended with φ_{extend}, demanding that the safety is increased by this operation.

We use DC formulae here to define timed constraints on the unknown parts *FAR* and *REC*. F_{FAR} demands that the safety cannot decrease below 0 within *CT* time units if *FAR* is entered with a safety distance large enough. Additionally, the phase is to be left after *CT* time units. And finally, F_{REC} demands that if it is entered with a safe *sf* value, then it remains safe.

The safety property, that is to be guaranteed, is that *sf* is never below or equal to 0, specified by the DC formula

$$\neg\Diamond(\lceil sf \leq 0 \rceil).$$

3.2. Properties of Extended CSP

As our semantics from Sect. 3.1.1 enriches standard CSP trace semantics by data constraints, one may ask whether important properties of the trace semantics carry over to our extension. One such property is that the trace semantics is continuous with respect to the $\subseteq$-order—a property that is useful when examining fixed points of recursive CSP expressions (cf. Sect. 4.4). It turns out that we can lift most of the CSP operators to a corresponding relation on interpretation level. To that, we apply a result of [Ros98] to conclude continuity of the operators. Unfortunately, this does not hold for the parallel composition operator, which is due to the presence of shared data. Anyhow, we first show the continuity for the remaining operators and discuss the problematic parallel composition afterwards.

3.2.1. Continuous CSP Operators

To show the continuity of CSP operators, we express all operators in terms of relations on interpretations such that the set of all interpretations of a process $P \oplus Q$ can be computed by applying the relations to the interpretation set of the sub-processes P and Q. For every binary operator $\oplus$ four relations are necessary: $[\oplus]_1$, mapping interpretations of the first participating process to output interpretations, $[\oplus]_2$ mapping interpretations of the second process to output interpretations, a ternary relation $[\oplus]_{1,2}$, describing the output interpretations for the case that both processes are active, and finally the set of output interpretations $[\oplus]_\bullet$ for the case that no process is active. With these relations the set of interpretations can be —identically to

[Ros98]—computed by lifting all operators to the interpretation level:

$$\begin{aligned}
[\![P \oplus Q]\!]\mathcal{M} &= [\![P]\!]\mathcal{M} \oplus_I [\![Q]\!]\mathcal{M} \\
S \oplus_I R &= \{u \mid \exists\, s \in S, t \in R : (s,t,u) \in [\oplus]_{1,2}\} \cup \\
&\quad \{u \mid \exists\, s \in S : (s,u) \in [\oplus]_1\} \cup \\
&\quad \{u \mid \exists\, t \in R : (t,u) \in [\oplus]_2\} \cup \\
&\quad [\oplus]_\bullet
\end{aligned} \tag{3.2}$$

For the unary operator $a \to P$ only $[\cdot]_1$ and $[\cdot]_\bullet$ are needed, for basic processes and unknown processes only $[\cdot]_\bullet$.

The relations for our process semantics are computed as follows[4].

$$\begin{aligned}
[\texttt{Stop}]_\bullet &= \{\mathcal{I} \mid \mathcal{I} = \langle \mathcal{M} \rangle\} \\
[\texttt{Skip}]_\bullet &= \{\mathcal{I} \mid \mathcal{I} = \langle \mathcal{M}, \checkmark \rangle\} \\
[a \bullet \varphi \to]_\bullet &= \{\mathcal{I} \mid \mathcal{I} = \langle \mathcal{M} \rangle\} \\
[a \bullet \varphi \to]_1 &= \{(\mathcal{I}_1, \mathcal{I}_2) \mid \mathcal{I}_1 = \langle \mathcal{M}, \dots \rangle, \mathcal{I}_2 = \langle \overline{\mathcal{M}}, a, \mathcal{M} \rangle \frown \mathcal{I}_1 \text{ s.t.} \\
&\qquad \overline{\mathcal{M}} \cup \mathcal{M}' \models \varphi \text{ and} \,\forall\, c \in \mathit{Const} : \mathcal{M}(c) = \overline{\mathcal{M}}(c)\} \\
[\Box]_{1,2} &= \{(\mathcal{I}, \langle \mathcal{M} \rangle, \mathcal{I}), (\langle \mathcal{M} \rangle, \mathcal{I}, \mathcal{I})\} \\
[\sqcap]_\bullet &= \{\langle \mathcal{M} \rangle\} \\
[\sqcap]_1 &= \{(\mathcal{I}, \mathcal{I})\} \\
[\sqcap]_2 &= \{(\mathcal{I}, \mathcal{I})\} \\
[\fatsemi]_1 &= \{(\mathcal{I}, \mathcal{I}) \mid \mathcal{I} \text{ does not terminate}\} \\
[\fatsemi]_{1,2} &= \{(\langle \mathcal{I}_1 \rangle \frown \langle \mathcal{M}, \checkmark \rangle, \mathcal{I}_2, \mathcal{I}) \mid \mathcal{I} = \mathcal{I}_1 \frown \mathcal{I}_2\} \\
[\texttt{Proc}_{\backslash A,V}]_\bullet &= \{\mathcal{I} \mid \mathcal{I} = \langle \mathcal{M}_0, a_1, \mathcal{M}_1, \dots, a_n, \mathcal{M}_n \rangle \text{ with} \\
&\qquad \forall\, i \in 0..n : a_i \notin A \,\wedge \\
&\qquad \forall\, v \in V, \forall\, i,j \in 0..n : \mathcal{M}_i(v) = \mathcal{M}_j(v)\} \\
[\texttt{Proc}^{\infty}_{\backslash A,V}]_\bullet &= \{\mathcal{I} \mid \mathcal{I} = \langle \mathcal{M}_0, a_1, \mathcal{M}_1, \dots, a_n, \mathcal{M}_n \rangle \text{ with } a_n \neq \checkmark \,\wedge \\
&\qquad \forall\, i \in 0..n : a_i \notin A \,\wedge \\
&\qquad \forall\, v \in V, \forall\, i,j \in 0..n : \mathcal{M}_i(v) = \mathcal{M}_j(v)\} \\
[\texttt{Proc}_{\backslash A,V} \bullet F]_\bullet &= \{\mathcal{I} \in [\![F]\!] \cap [\texttt{Proc}_{\backslash A,V}]_\bullet\} \\
[\texttt{Proc}^{\infty}_{\backslash A,V} \bullet F]_\bullet &= \{\mathcal{I} \in [\![F]\!] \cap [\texttt{Proc}^{\infty}_{\backslash A,V}]_\bullet\} \\
[\backslash X]_1 &= \{(\mathcal{I}_1, \mathcal{I}_2) \mid \mathcal{I}_1 \text{ fits to run } \pi \text{ and } \mathcal{I}_2 \text{ fits to run } \pi \setminus X\} \\
[[R]]_1 &= \{(\mathcal{I}_1, \mathcal{I}_2) \mid \mathcal{I}_1 = \langle \mathcal{M}_0, a_1, \mathcal{M}_1, a_2, \dots \rangle \,\wedge \\
&\qquad \mathcal{I}_2 = \langle \mathcal{M}_0, b_1, \mathcal{M}_1, b_2, \dots \rangle \text{ with } \forall\, i : a_i R b_i\}
\end{aligned}$$

[4]If a relation is not given for an operator it is empty. To avoid cluttering up the definitions we omit the quantification over the model $\mathcal{M}$: if not stated else the definitions hold for arbitrary models $\mathcal{M} : \mathit{Model}$.

Here, the run $\pi \setminus X$ is the restriction of π to events not in X.

These definitions of relations for every CSP operator allows us to convey the following theorem about the continuity of the operators.

Theorem 3.2.1 (Continuity of CSP operators in trace semantics)
All CSP operators of our CSP extension with data and unknowns, except for parallel composition, are continuous with respect to the underlying interpretation semantics and the $\subseteq$-order. Continuity here means that every operator on interpretation level $\oplus_I$ (with one argument fixed) applied to the least upper bound of a non-empty directed[5] set S of interpretations[6], written $\bigsqcup S$, is equal to the least upper bound of the set where $\oplus_I$ is applied to each interpretation in S, i.e.,

$$X \oplus_I (\bigsqcup S) = \bigsqcup \{X \oplus_I Y \mid Y \in S\}.$$

Proof. In [Ros98] it is proven that every operator that can be lifted to its semantical domain using a family of relations, as in equation (3.2), is continuous in each argument with respect to the $\subseteq$-order. ☐

For this notion of continuity, every continuous operator is also monotone.

An important consequence of this result is that recursive processes defined over these continuous operators can be identified with its unique fixed point [Ros98], which particularly facilitates the definition of an induction-based proof rule in Sect. 4.3.6.

3.2.2. Discontinuity of Parallel Composition

When directly conveying the definitions for parallel composition of [Ros98] from traces to interpretations, we get the relation

$$[\|_A]_{1,2} = \{(\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}) \mid \mathcal{I} \in \mathcal{I}_1 \|_A \mathcal{I}_2\}$$

with

$$\begin{aligned}
\mathcal{I}_1 \|_A \mathcal{I}_2 &= \mathcal{I}_2 \|_A \mathcal{I}_1 \\
\langle \mathcal{M} \rangle \|_A \langle \mathcal{M} \rangle &= \{\langle \mathcal{M} \rangle\} \\
\langle \mathcal{M} \rangle \|_A \langle \mathcal{M}, a, \overline{\mathcal{M}} \rangle &= \{\langle \mathcal{M} \rangle\} \\
\langle \mathcal{M} \rangle \|_A \langle \mathcal{M}, b, \overline{\mathcal{M}} \rangle &= \{\langle \mathcal{M}, b, \overline{\mathcal{M}} \rangle\} \\
\langle \mathcal{M}_1, a, \mathcal{M}_2 \rangle ^\frown \mathcal{I}_1 \|_A \langle \overline{\mathcal{M}}_1, b, \overline{\mathcal{M}}_2 \rangle ^\frown \mathcal{I}_2 &= \{\langle \overline{\mathcal{M}}_1, b, \overline{\mathcal{M}}_2 \rangle ^\frown \mathcal{I} \mid \\
&\qquad \mathcal{I} \in \langle \mathcal{M}_1, a, \mathcal{M}_2 \rangle ^\frown \mathcal{I}_1 \|_A \mathcal{I}_2\} \\
\langle \mathcal{M}_1, a, \mathcal{M}_2 \rangle ^\frown \mathcal{I}_1 \|_A \langle \mathcal{M}_1, a, \mathcal{M}_2 \rangle ^\frown \mathcal{I}_2 &= \{\langle \mathcal{M}_1, a, \mathcal{M}_2 \rangle ^\frown \mathcal{I} \mid \mathcal{I} \in \mathcal{I}_1 \|_A \mathcal{I}_2\}
\end{aligned}$$

[5] A set S is directed if each finite subset has an upper bound in S.

[6] The least upper bound exists, because the set of interpretations is a complete lattice with respect to the $\subseteq$-order.

$$\langle \mathcal{M}_1, a_1, \mathcal{M}_2\rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{M}_1, a_2, \overline{\mathcal{M}}_2\rangle \frown \mathcal{I}_2 = \{\langle \mathcal{M}_1\rangle\}$$

$$\begin{aligned}\langle \mathcal{M}_1, b_1, \mathcal{M}_2\rangle \frown \mathcal{I}_1 \parallel_A \langle \overline{\mathcal{M}}_1, b_2, \overline{\mathcal{M}}_2\rangle \frown \mathcal{I}_2 = &\{\langle \overline{\mathcal{M}}_1, b_2, \overline{\mathcal{M}}_2\rangle \frown \mathcal{I} \mid \\ &\ \mathcal{I} \in \langle \mathcal{M}_1, b_1, \mathcal{M}_2\rangle \frown \mathcal{I}_1 \parallel_A \mathcal{I}_2\} \cup \\ &\{\langle \mathcal{M}_1, b_1, \mathcal{M}_2\rangle \frown \mathcal{I} \mid \\ &\ \mathcal{I} \in \mathcal{I}_1 \parallel_A \langle \overline{\mathcal{M}}_1, b_2, \overline{\mathcal{M}}_2\rangle \frown \mathcal{I}_2\}\end{aligned}$$

for $a, a_1, a_2 \in A$ with $a_1 \neq a_2$ and $b, b_1, b_2 \notin A$. The trace set is empty for all remaining cases. Note that $\frown$ has a higher precedence than $\parallel$. This definition of trace synchronisation reveals a characteristic of the CSP semantics with data constraints: parallel components synchronise on events as specified in the synchronisation alphabet and on the data constraints if a common transition step is possible, but if one component executes a step single-handedly it may change the system variables arbitrarily (according to its constraints). Thus, this definition is actually only correct for processes synchronising on all events and for data-independent parallel components, i.e., components that do not read symbols that are changed by the other component. The following example demonstrates this issue.

Example 3.2.2. An example shows why continuity and even monotonicity for parallel composition with shared data fails. Consider the processes

$$\begin{aligned}P_1 &\overset{c}{=} (a \bullet x' = 1 \land y' = 1) \to \\ &\quad \big((b_1 \bullet x = 1 \land y' = 1 \to \texttt{Skip}) \,\Box\, (b_2 \bullet x = 0 \land y' = 0 \to \texttt{Skip})\big) \\ P_2 &\overset{c}{=} (c \bullet x' = 0 \land y' = y) \to \texttt{Skip}.\end{aligned}$$

With $\mathcal{M}_0 \models y = 1$, the interpretations of $[\![P_1]\!]\mathcal{M}_0$ are of the shape

$$\langle \mathcal{M}_0, a, \mathcal{M}_1, b_1, \mathcal{M}_2\rangle,$$

s.t. $\mathcal{M}_i \models y = 1$ for $i \in 1..2$. All interpretations of $[\![P_2]\!]\mathcal{M}_0$ have the shape

$$\langle \mathcal{M}_0, c, \mathcal{M}_3\rangle,$$

with $\mathcal{M}_3 \models x = 0$ and $\mathcal{M}_3 \models y = 1$. Particularly, all interpretations of P_1 and P_2 satisfy $y = 1$ for all its models, and additionally, the branch of P_1 with b_2 is never activated. Thus, b_2 does not occur in the set of interpretations for P_1.

But in the parallel composition of P_1 and P_2, $[\![P_1 \parallel P_2]\!]\mathcal{M}_0$, the interpretation

$$\langle \mathcal{M}_0, a, \mathcal{M}_1, c, \mathcal{M}_3, b_2, \mathcal{M}_4\rangle$$

with $\mathcal{M}_4 \models y = 0$ occurs, because the b_2 branch is now activated since P_2 sets $x' = 0$. For this reason, the set $[\![P_1 \parallel P_2]\!]\mathcal{M}_0$ contains interpretations that cannot be derived from the semantics of its parts $[\![P_1]\!]\mathcal{M}_0$ and $[\![P_2]\!]\mathcal{M}_0$. The parallel composition operator is neither continuous nor monotone for the interpretations semantics.

For this reason, our interpretation-based semantics has the weakness that parallel compositions cannot directly be used under recursion because, in this case, the recursion does not need to have a unique fixed point anymore. For simple parallel compositions without unknown processes this is actually not relevant for our proof system, because parallel composition is anyway resolved by expansion into a process over choices (Def. 3.3.1). This is possible, since the interpretation semantics of a process is computed by first deriving a trace from the LTS of a process and, then, the data-part is considered in a second step. In this second step, the information about the potential structure (like the second branch of the P_1 choice in Example 3.2.2) of the process gets lost. By translating the process into an equivalent representation without parallel composition before computing its interpretations, we can circumvent this problem.

With parallel composition over constrained unknowns, this is not possible anymore because we do not have an LTS representation of such a process and cannot translate it into a process without parallel composition. Therefore, we investigate parallel compositions over unknowns in Chap. 6 and present an alternative semantics for processes with data in Sect. 6.1.5, which is more complex but does not have the identified problems. A further approach that will presumably not have these problems is to use a parallel composition operator that synchronises also on data similarly to PEA. But this does not reflect the behaviour of systems that can access common variables (like this is also the case for processes *within* CSP-OZ-DC class specifications), that are the application domain of this work. Hence, we do not consider this approach here.

3.3. Normal Forms

Before introducing our dynamic logic extension dCSP for CSP processes with data and sequent calculus rules, we need a further utility that helps us handle parallel composition and interleaving in the process terms occurring in dCSP expressions: a guarded normal form for CSP expressions.

Definition 3.3.1 (Guarded Normal Form)
A CSP process (with data) P is in Guarded Normal Form (GNF) *if it is of the shape*

$$P ::= \Box\, a_i : \Sigma \bullet a_i \to P_i \mid \Box\, a_i : \Sigma \bullet a_i \to P_i \,\Box\, \texttt{Skip} \mid \texttt{Skip} \mid \texttt{Stop} \mid X$$
$$\mid \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \mid \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \boxtimes P \mid P \boxtimes \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F,$$

where $a_i \neq a_j$ for $i \neq j$, $\Sigma \subseteq \mathbb{U}_{Events}$, and $\boxtimes \in \{\|, \Box, \text{;}\}$. The process call X is declared by $X \stackrel{c}{=} Q$ where Q is a choice, $\Box\, a_i : \Sigma \bullet a_i \to P_i$ or $\Box\, a_i : \Sigma \bullet a_i \to P_i \,\Box\, \texttt{Skip}$, that has to be again a process in GNF.

Note that the events a_i can be constrained events $a_i \bullet \varphi$, but we omit the constraints in the definition, because they are of no relevance here.

An unknown process $\texttt{Proc}^{(\infty)}_{\backslash A,V}$ is considered to be in GNF, because we apply them like basic processes that cannot be further decomposed. Thus, we also need to list unknown processes in all possible operator contexts in the definition.

We directly affiliate the following theorem:

Theorem 3.3.2
Every CSP process is trace-equivalent to a process in guarded normal form.

Proof. Let P be a CSP process with data. The proof is by structural induction over P. For the basic cases $P = \texttt{Stop}, P = \texttt{Skip}$, and $P = \texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F$ the proposition follows immediately. For $P = X$, we need to examine the declaring process. So, our induction hypothesis is that the proposition holds already for all sub-processes of P. We examine the possible top-level operators of P (we omit all constraints of events since they are not relevant for the proof):

$P = a \to Q$: Then P is of the desired shape $\Box_{i \in 1..1}\, a \to Q$.

$P = P_1 \sqcap P_2$: In trace semantics also $P \equiv P_1 \Box P_2$.

$P = P_1 \Box P_2$: We show for processes P_1 and P_2 in GNF by induction over the sum of the maximal nesting depth of the replicated choice operator $\Box\, a_i : \Sigma \bullet a_i \to Q_i$ in P_1 and P_2 that $P_1 \Box P_2$ is also in GNF.

$P_1 = P_2 = \texttt{Skip}$: Then, $P \equiv \texttt{Skip}$.

$P_1 = \texttt{Stop}$: Then, $P \equiv P_2$.

$P_1 = \texttt{Skip}, P_2 = \Box\, a_i : \Sigma \bullet a_i \to Q_i$: Then, P is already of the desired shape.

$P_1 = \texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F$: Then, P is already of the desired shape.

$P_1 = \texttt{Skip}, P_2 = \Box\, a_i : \Sigma \bullet a_i \to Q_i \Box \texttt{Skip}$: Then, $P \equiv P_2$.

$P_1 = \Box\, a_i : \Sigma_1 \bullet a_i \to Q_i, P_2 = \Box\, b_j : \Sigma_2 \bullet b_j \to \overline{Q}_j$: Let $a_i = b_i$ for $i \in 1..n$. We build up a new external choice of the desired shape:

$$P \equiv \Box_{i \in 1..n}\, a_i \to \mathit{GNF}(Q_i \Box \overline{Q}_i) \Box \Box_{i>n}\, a_i \to Q_i \Box \Box_{i>n}\, b_i \to \overline{Q}_i, \tag{3.3}$$

in which $\mathit{GNF}(Q_i \Box \overline{Q}_i)$ is a GNF process equivalent to $Q_i \Box \overline{Q}_i$. Such a process exists, because the induction hypothesis is applicable, since the sum of the maximal choice nesting depth in $Q_i \Box \overline{Q}_i$ is lower than for P.

If a Skip occurs in the choices of P_1 or P_2 then it also occurs in (3.3). The symmetric cases are omitted.

$P = P_1 \parallel_A P_2$: We show for processes P_1 and P_2 in GNF by induction over the sum of the maximal nesting depth of the replicated choice operator $\Box\, a_i : \Sigma \bullet a_i \to Q_i$ in P_1 and P_2 that $P_1 \parallel_A P_2$ has also an equivalent process in GNF.

$P_1 = \texttt{Stop}, P_2 = \texttt{Stop}$: Then, $P \equiv P_1 \parallel_A P_2 \equiv \texttt{Stop}$ is in GNF.

$P_1 = \texttt{Skip}, P_2 = \texttt{Skip}$: Then, $P \equiv P_1 \parallel_A P_2 \equiv \texttt{Skip}$ is in GNF.

$P_1 = \texttt{Skip}, P_2 = \texttt{Stop}$: Then, $P \equiv P_1 \parallel_A P_2 \equiv \texttt{Stop}$ is in GNF.

$P_1 = \Box\, a_i : \Sigma_1 \bullet a_i \to Q_i, P_2 = \Box\, b_j : \Sigma_2 \bullet b_j \to \overline{Q}_j$: Then,

$$\begin{aligned} P \equiv\; & (\Box\, a_i : A \cap \Sigma_1 \cap \Sigma_2 \bullet a_i \to Q_i \parallel_A \overline{Q}_i) \\ & \Box\, (\Box\, a_i : \Sigma_1 \setminus (A \cup \Sigma_2) \bullet a_i \to Q_i \parallel_A P_2) \\ & \Box\, (\Box\, a_i : \Sigma_2 \setminus (A \cup \Sigma_1) \bullet a_i \to P_1 \parallel_A \overline{Q}_i) \\ & \Box\, (\Box\, a_i : \Sigma_1 \cap \Sigma_2 \cap \overline{A} \bullet a_i \to (P_1 \parallel_A \overline{Q}_i) \Box (Q_i \parallel_A P_2)), \end{aligned}$$

where we can apply the induction hypothesis to $Q_i \parallel_A \overline{Q}_i$, $Q_i \parallel_A P_2$, and $P_1 \parallel_A \overline{Q}_i$ because the maximal choice nesting depth is reduced in at least one component. Thus, they can all be represented in GNF. For $(P_1 \parallel_A \overline{Q}_i) \Box (Q_i \parallel_A P_2)$, we apply the induction hypothesis to each of the choice operands and get that $P_1 \parallel_A \overline{Q}_i$ and $Q_i \parallel_A P_2$ can be represented in GNF. We apply the insight from the previous case that choices of GNF processes can again be represented in GNF to get that $(P_1 \parallel_A \overline{Q}_i) \Box (Q_i \parallel_A P_2)$ can be represented in GNF and, thus, P is also in GNF.

$P_1 = \Box\, a_i : \Sigma_1 \bullet a_i \to Q_i, P_2 = \texttt{Skip}$: Then, $P \equiv \Box\, a_i : \Sigma_1 \setminus A \bullet a_i \to Q_i$. If $\Sigma_1 \setminus A$ is empty then $P \equiv \mathit{Stop}$.

$P_1 = \Box\, a_i : \Sigma_1 \bullet a_i \to Q_i, P_2 = \texttt{Stop}$: Then, $P \equiv \Box\, a_i : \Sigma_1 \setminus A \bullet a_i \to (Q_i \parallel \texttt{Stop})$. By applying the induction hypothesis, $Q_i \parallel \texttt{Stop}$ can be represented in GNF. If $\Sigma_1 \setminus A$ is empty, then $P \equiv \mathit{Stop}$.

The symmetric cases are omitted again.

$P = P_1 \fatsemi P_2$: If $P_1 = \texttt{Stop}$, then $P \equiv \texttt{Stop}$; if $P_1 = \texttt{Skip}$, then $P \equiv P_2$; and if $P_1 = \Box\, a_i : \Sigma_1 \bullet a_i \to Q_i$, then $P \equiv \Box\, a_i : \Sigma_1 \bullet a_i \to (Q_i \fatsemi P_2)$. All of these cases can be converted to GNF by applying the induction hypothesis.

□

Since the interpretations of a process are computed by first calculating the traces of the process and then translating the traces into interpretations, we can directly infer the following theorem about the interpretations of GNF processes.

Corollary 3.3.3
Every process is interpretation-equivalent to a process in guarded normal form.

Another helpful representation of CSP processes with data is the *unknown process normal form*, which partitions a CSP process into its defined process parts and its unknown parts.

Definition 3.3.4 (Unknown process normal form)
A CSP process P is in unknown process normal form*, denoted* `Proc` normal form*, if it is of the shape*

$$\begin{aligned} P &\overset{c}{=} Q \\ X_1 &\overset{c}{=} \texttt{Proc}^{(\infty)}_{\backslash A_1, V_1} \bullet \varphi_1 \\ &\vdots \\ X_n &\overset{c}{=} \texttt{Proc}^{(\infty)}_{\backslash A_n, V_n} \bullet \varphi_n, \end{aligned}$$

where Q is a process without $\texttt{Proc}^{(\infty)}$ processes that may contain references to $X_1, \ldots,$ X_n but to no other processes. The process Q is called characteristic process *of P. We also use this normal form for unconstrained processes $\texttt{Proc}^{(\infty)}$, which means that all φ_i are true.*

In the proof of Thm. 3.3.6, we need an operator to rename process identifiers in CSP expressions:

Definition 3.3.5 (Renaming of process identifiers)
Given a CSP process P, the renaming of a process identifier *Y into a fresh identifier X, written $P[X/Y]$, is defined as follows:*

$$P[X/Y] := \begin{cases} X & \text{if } P = Y \\ a \to Q[X/Y] & \text{if } P = a \to Q \\ Q_1[X/Y] \; op \; Q_2[X/Y] & \text{if } P = Q_1 \; op \; Q_2 \\ \mu \, Z \bullet F(Z)[X/Y] & \text{if } P = \mu \, Z \bullet F(Z) \\ \mu \, X \bullet F(X) & \text{if } P = \mu \, Y \bullet F(Y) \\ P & \text{otherwise,} \end{cases}$$

where $op \in \{\Box, \|, |||, \sqcap, ;\}$ and $Z \neq Y$; X, Y and Z denote process identifiers and P, Q and Q_i processes.

Theorem 3.3.6
Let P be a CSP process with data containing references to unknown processes, then P can be equivalently represented in `Proc` *normal form.*

Proof. In a first step, we remove all references to process identifiers in the definition of P. Let P be represented by a process equation system

$$\begin{aligned} X &= P \\ X_1 &\overset{c}{=} P_1 \\ &\vdots \\ X_n &\overset{c}{=} P_n, \end{aligned}$$

such that all P_i may contain references to $X, X_1, \ldots, X_n$.

We now replace inductively all occurrences of process identifiers X_i in P. Starting by index $i = 1$, every ocurrence of X_1 in P and P_i with $i > 1$ is replaced by $\mu\, Y_1 \bullet P_1[Y_1/X_1]$, where Y_1 is a fresh identifier. Afterwards, let the resulting process equation system be

$$X = P'$$
$$X_2 \overset{c}{=} P'_2$$
$$\vdots$$
$$X_n \overset{c}{=} P'_n.$$

Then, $P \equiv P'$ and P' does not contain identifier X_1 anymore. We iterate this procedure until no references to other process equations are left in P'. This procedure terminates because we reduce the number of process equations in every iteration.

We give now a proof that P' can be represented in `Proc` normal form by induction over the structure of P': for the base cases, where P' equals `Stop` or `Skip`, P' is already in `Proc` normal form. For the case that P' equals $\mathtt{Proc}^{\infty}_{\backslash A, V} \bullet \varphi$, we introduce a new process equation $Z_1 \overset{c}{=} \mathtt{Proc}^{\infty}_{\backslash A, V} \bullet \varphi$, and $P'' \overset{c}{=} Z_1$ is equivalent to P' and in `Proc` normal form.

For the induction step, we consider $P \overset{c}{=} P_1\, op P_2$ with $op \in \{\Box, \|, |||, \sqcap, \fatsemi\}$, where we assume that P_1 and P_2 are already in `Proc` normal form with characteristic processes $\overline{P_1}$ and $\overline{P_2}$ as well as process identifiers representing unknown processes $Z_1, \ldots, Z_n$ for P_1 and $Z_{n+1}, \ldots, Z_m$ for P_2. Then, P' can equivalently be represented by $P'' \overset{c}{=} \overline{P_1}\, op \overline{P_2}$, which is again in normal form with identifiers $Z_1, \ldots, Z_m$ standing for unknown processes. The prefix case $P' \overset{c}{=} a \rightarrow P_1$ is analog. Since by construction P' contains no process references X_i, the proof is finished. □

3.4. Discussion

3.4.1. Parametric Systems

Our CSP extension by data and unknown processes enables us to specify architectures of parametric systems: Firstly, we can use global data parameters as declared in the signature of a specification. These parameters have fixed but arbitrary values for every system run. Secondly, the unknown processes give a parametric view to process components that are not fixed but instead represent a class of concret processes. In doing so, we provide a general formalism that is on the one hand flexible enough to express behavioural protocols with a large degree of freedom, and on the other hand, integrates well with combined specification formalisms based on CSP [MD98, Fis00, WC01, Süh02, Hoe06, SLD08]. So, our goal was not to introduce a further combination

of CSP with data as a replacement for existing formalisms but to provide a notation for VAs that can be used in combination with these formalisms.

CSP with data extends standard CSP in a direct way by integrating state changes directly into processes, by which the formalism enables a concise presentation of heterogenous systems involving data and time without complex syntactic constructs as in other combined formalisms. In particular, it turned out that direct usage of a combined formalism with complex syntactic structures like CSP-OZ-DC is not appropriate for a proof rule approach because of the combination of several languages in an object-oriented structure. Nevertheless, it is not the objective of the CSP extension to provide a developer-friendly syntax for the modelling of complex systems, because that has been the goal of the aforementioned combined approaches. Instead it extracts the basic elements of a lot of combined formalisms, processes and data changes, and thus can be used in combination with them like presented in this work for the Verification Architecture approach.

3.4.2. Semantics

The chosen semantics of our CSP extension from Sect. 3.1.1 directly reflects the trace semantics of CSP. By this, its integration of the event traces of a process with data changes are very similar to the CSP-OZ-DC approach (Sect. 2.2.2) and nevertheless general enough for applications beyond CSP-OZ-DC.

On the downside, this leads to a discontinuous parallel composition operator, which has been examined in Sect. 3.2.2. With the possibility to reformulate parallel compositions into choices of processes, this causes no problems for the basic proof rule approach presented in the next chapter, as long as processes without parallelism over unknown parts are considered. We address the problems arising from the discontinuity of the parallel composition operator, in particular with parallelism over unknown parts, in Sect. 6.1.

Apparently, one difference in the semantics of CSP-OZ-DC and eCSP is the compositionality of CSP-OZ-DC and PEA (Sect. 2.3.2 and Thm. 2.3.8) that does not hold for eCSP, even though the semantics of CSP-OZ-DC classes and eCSP processes are defined in a similar way. The reason for this maybe unexpected difference is that CSP-OZ-DC compositionality only holds with respect to CSP-OZ-DC *classes* (and for its CSP, OZ, DC parts) but not *within* the CSP process of a CSP-OZ-DC class. That is, the compositionality result does not hold for the components of a CSP process: a process $\texttt{main} \stackrel{c}{=} P_1 \parallel P_2$ can not be verified by translating P_1 and P_2 into PEA and checking properties of them separately because P_1 and P_2 may fire events changing the state space of the system in an asynchronous way. The state change caused by one component does not need to be reflected by the other component, which is the basis for the compositionality of CSP-OZ-DC verification. So, the semantics of CSP with data fits to the semantics of CSP processes within CSP-OZ-DC classes, by which the CSP extension can be used to analyse and decompose processes of single CSP-OZ-DC classes (using the verification approach of the following chapters).

3.4.3. Related work

We already mentioned some combined formalisms in Sect. 2.2.3, and as CSP with data is not intended to replace existing combined formalisms with a more complex syntactical structure, we here focus on approaches that integrate data or unknown components in a similar way.

CSP with data. The *Process Analysis Toolkit (PAT)* is an analysis and verification framework for CSP# [SLD08, SLDC09]. The idea is to specify data aspects in sequential, terminating programs in terms of an imperative programming language. These programs are used as atomic CSP events. Thus, CSP# is similar to our CSP processes with data constraints with a different focus: they use a simple procedural low-level programming language (in C# syntax), whereas we here use a general CSP extension with declarative logical constraints (generally over arbitrary sorts without a fixed syntax). Events corresponding to data changes cannot be used for synchronisations in PAT. Due to the generality of our CSP extension, CSP# programs can be carried over to CSP with data constraints. PAT supports complex data structures but currently only with a finite data domain. In [Liu09, SLDP09] this approach is extended towards Timed CSP by integrating fixed real-time constraints to specify deadlocks, timed interrupts and time-outs.

Unknown components. The early work of Larsen and Xinxin [LX91] introduces a concept similar to the unknown processes of this work. They define *context systems* that are partial designs of systems that can be instantiated with concrete processes. The work is not bound to a specific process algebra but gives a general operational semantics for contexts. The unknown processes of eCSP can be modelled by these contexts. Context systems are designed to generally reflect arbitrary process algebras for theoretical analysis, and thus are less suited for modelling of concrete systems. Hence, we preferred the explicit representation of CSP processes with unknown parts, but the results of Larsen and Xinxin carry over to our approach. They examine how to generate compositional formulae in Hennessy-Milner logic [HM85] that have to be satisfied by the unknown components of the partial design in order to satisfy the entire specification. The authors consider the generation of assumptions for a specific logic, while we use a fixed process algebra but allow arbitrary real-time logics for the assumptions. In [LX91] no real-time aspects are examined and neither are data aspects. They do not give a proof system to establish properties of given partial designs but suggest to use step-wise refinement to derive correct implementations of an initial partial design.

Another related approach developed by D'Errico and Loreti that is based on Hennessy-Milner logic and assumption-guarantee verification can be found in [DL09, DL10]. It presents a sound and complete tableau-based proof system to reason about CCS processes with assumptions on the environment. These assumptions as well as the properties to be proven for the overall system are specified in (different) dialects

of Hennessy-Milner logic. There are several differences to our approach: neither real-time properties, data constraints, nor combined specifications are considered. The assumptions on the environment can be compared to our constrained unknowns: the unknown processes of eCSP are explicitly represented as process expressions, and thus they can be flexibly used everywhere within processes. On the contrary, the environment assumptions of D'Errico and Loreti are always composed in parallel to the system process. Both approaches have in common that they are property-preserving for instantiations of the unknown parts.

4 A Sequent Calculus for Verification Architectures

Everything changes, but nothing is truly lost.

(Dream, in The Wake, *Neil Gaiman)*

4.1. Dynamic Logic over CSP Processes with Data 66
4.2. Sequent Calculus 68
4.3. Proof Rules 70
4.3.1. Structural Rules 70
4.3.2. Propositional Rules 70
4.3.3. First-Order Rules 71
4.3.4. Symbolic Execution of dCSP Formulae 72
4.3.5. Symbolic Execution of dCSP Specifications with Unknown Processes 75
4.3.6. Induction Rules 77
4.3.7. Auxiliary dCSP Rules 79
4.4. Soundness of the Calculus 80
4.5. Embedding of a Real-Time Logic 93
4.5.1. Checking the Side-Conditions for the Box Operator 93
4.5.2. Checking the Side-Conditions for the Diamond Operator . . 95
4.6. Discussion 99
4.6.1. Discussion of the VA Approach 99
4.6.2. Related Work 100

In order to show safety properties of CSP processes with unknown processes and additional temporal constraints, we introduce a sequent calculus in this chapter. It turned out (cf. Chap. 9) that the straightforward use of model checking to verify VAs is not working due to the potentially large number of parallel local assumptions for unknown components. Thus, different verification approaches are needed, and we suggest to apply a rule-based approach here. Instead of defining novel free-style proof rules, we decided to integrate our CSP dialect with Dynamic Logic [Har79, Har84, HKT00] and to give proof rules in a sequent-style calculus [Gen35]. By this, the advantages of Dynamic Logic for program verification carry over to our approach: Dynamic Logic is—in opposite to Hoare logic [Hoa69]—closed under first-order operations, we can use existing compositional proof rules for the first-order part of our logic, and we also benefit from existing tools [HHRS86, BHS07, PQ08, PQR09] that prove Dynamic Logic to be well-suited for automatisation.

4.1. Dynamic Logic over CSP Processes with Data

We now define dCSP, an extension of Dynamic Logic that we use to verify Verification Architectures given by eCSP processes. The main idea is to use eCSP processes instead of programs within the box operator $[\,\cdot\,]$ and the diamond operator $\langle\,\cdot\,\rangle$.

Definition 4.1.1 (Syntax of dCSP formulae)
We consider a signature $\Sigma = (Sort, SysVar, Const, Var)$ and define the set $Form_{dCSP}$ of dCSP formulae *inductively:*

$$\begin{array}{ll}
\textit{if } p \in Form_\Sigma, p : S_1 \times \cdots \times S_n \to \mathbb{B} \textit{ and} & \\
\quad \theta_1, \ldots, \theta_n \in Term_\Sigma, \theta_1 : S_1, \ldots, \theta_n : S_n & \textit{then } p(\theta_1, \ldots, \theta_n) \in Form_{dCSP} \\
\textit{if } \delta_1, \delta_2 \in Form_{dCSP} & \textit{then } (\neg\delta_1), (\delta_1 \wedge \delta_2) \in Form_{dCSP} \\
\textit{if } \delta \in Form_{dCSP}, x \in Var & \textit{then } (\forall\, x \bullet \delta), (\exists\, x \bullet \delta) \in Form_{dCSP} \\
\textit{if } \delta \in Form_{dCSP}, P \textit{ a CSP process} & \textit{then } ([P]\delta), (\langle P\rangle\delta) \in Form_{dCSP} \\
\textit{if } \varphi \in Form_{dCSP}, P \textit{ a CSP process} & \textit{then } ([P]\Box\varphi), (\langle P\rangle\Diamond\varphi) \in Form_{dCSP}
\end{array}$$

We use $\langle\!\![P]\!\!\rangle\gamma$ if a formula or rule applies to both $[P]\gamma$ and $\langle P\rangle\gamma$.

The symbol $\overline{x}$ denotes vectors of variables and $\exists\,\overline{x} \bullet \varphi$ (and analogously for $\forall$) denotes quantifications over vectors of variables. We use the convention that δ and ϵ are dCSP formulae, i.e. particularly, they never begin with a path operator $\Diamond$ or $\Box$, whereas γ always represents a dCSP formula δ or a *path formula* $\Box\delta$ or $\Diamond\delta$.

So this definition allows first-order combinations over the box and the diamond operator that can be arbitrarily nested. Thus, constructs like

$$([P_1]\langle P_2\rangle\Diamond[P_3]\delta) \Rightarrow [Q]\Box\varphi$$

are in the syntax of dCSP. Not contained is direct nesting of path operators, i.e., $[P]\Box\Diamond\delta$, and constructs with alternations, $[Q]\Diamond\varphi$. The latter could easily be allowed in syntax and semantics of dCSP, but we do not have proof rules for those constructs, and therefore we omit them.

Before defining the semantics of dCSP formulae, we introduce the notion of a *terminating model*, which is the last, stable model in an interpretation of a terminating process.

Remark 4.1.2 (Handling of termination and deadlocks). On the semantical level, termination is represented by interpretations for finite untimed event sequences with ✓ as the last event (cf. Sect. 3.1.1). Like for normal events, the ✓-event is modelled by a Boolean variable that changes exactly at the point in time when the ✓-event occurs. Deadlocks do not have a semantical representation because of the underlying trace semantics.

Definition 4.1.3 (Terminating model)
An interpretation $\mathcal{I} : Time \to Model$ has a terminating model iff there is a point in time $t_0 \in Time$ at which the event ✓ occurs, and for every $t_i \in Time$ with $t_i > t_0$ for $i \in 1..2$

$$\mathcal{I}(t_1) = \mathcal{I}(t_2)$$

holds. Then, $\mathcal{I}(t_1)$ is called the terminating model.

We now define the semantics of dCSP formulae.

Definition 4.1.4 (Semantics of dCSP formulae)
The semantics of a dCSP term $f(\theta_1, \ldots, \theta_n)$ with sort S of f is a mapping $[\![\cdot]\!] : Model \to \mathcal{D}_S$ defined by

$$[\![f(\theta_1, \ldots, \theta_n)]\!]\mathcal{M} = f_{\mathcal{I}}([\![\theta_1]\!]\mathcal{M}, \ldots, [\![\theta_n]\!]\mathcal{M}).$$

The semantics of dCSP formulae is given by models $\mathcal{M} \in Model$:

$\mathcal{M} \models p(\theta_1, \ldots, \theta_n)$	*iff* $p_{\mathcal{I}}([\![\theta_1]\!]\mathcal{M}, \ldots, [\![\theta_n]\!]\mathcal{M})$
$\mathcal{M} \models \neg\gamma$	*iff* $\mathcal{M} \not\models \gamma$
$\mathcal{M} \models \gamma_1 \wedge \gamma_2$	*iff* $\mathcal{M} \models \gamma_1$ *and* $\mathcal{M} \models \gamma_2$
$\mathcal{M} \models \forall\, x \bullet \gamma$	*iff for all* $d \in \mathcal{D}_S$ *holds* $\mathcal{M}[x \mapsto d] \models \gamma$
$\mathcal{M} \models \exists\, x \bullet \gamma$	*iff there is a* $d \in \mathcal{D}_S$ *s.t.* $\mathcal{M}[x \mapsto d] \models \gamma$
$\mathcal{M} \models [P]\Box\delta$	*iff* $\mathcal{I} \models \Box\delta$ *holds for every interpretation* $\mathcal{I} \in [\![P]\!]\mathcal{M}$
$\mathcal{M} \models [P]\delta$	*iff* $\mathcal{M}' \models \delta$ *holds for every* $\mathcal{I} \in [\![P]\!]\mathcal{M}$ *with terminating model* $\mathcal{M}'$
$\mathcal{M} \models \langle P\rangle\Diamond\delta$	*iff* $\mathcal{I} \models \Diamond\delta$ *holds for some*

	interpretation $\mathcal{I} \in [\![P]\!]\mathcal{M}$
$\mathcal{M} \models \langle P \rangle \delta$	*iff* $\mathcal{M}' \models \delta$ *holds for some* $\mathcal{I} \in [\![P]\!]\mathcal{M}$ *with*
	terminating model $\mathcal{M}'$

Here, S *is the sort of variable* x. *The formula* $\Box\delta$ *holds for an interpretation* $\mathcal{I}$, *i.e.,* $\mathcal{I} \models \Box\delta$, *iff for all* $t \in Time \bullet \mathcal{I}(t) \models \delta$. *Analogously,* $\mathcal{I} \models \Diamond\delta$ *if there is a point in time* $t_0 \in Time$ *with* $\mathcal{I}(t_0) \models \delta$. *For the sake of completeness, we also define* $\mathcal{I} \models \delta$ *if* $\mathcal{I}$ *has a terminating model* $\mathcal{M}$ *with* $\mathcal{M} \models \delta$ *or* $\mathcal{I}$ *does not terminate. The set* $[\![\gamma]\!]$ *then denotes the set of all interpretations satisfying* γ, *defined by*

$$[\![\gamma]\!] := \{\mathcal{I} \mid \mathcal{I} \models \gamma\}.$$

4.2. Sequent Calculus

In this section, we define verification rules in a sequent calculus to prove validity of dCSP formulae.

Definition 4.2.1 (Sequent)
Given finite sets of formulae Δ *and* Γ, *we define the* sequent $\Delta \vdash \Gamma$ *as the abbreviation for the dCSP formula*

$$\bigwedge_{\varphi \in \Delta} \varphi \Rightarrow \bigvee_{\psi \in \Gamma} \psi.$$

The formulae Δ *on the left side of the sequent symbol* $\vdash$ *are called* antecedent*; the formulae* Γ *on the right side* succedent. *We also write* $\Delta, \varphi \vdash \psi, \Gamma$ *if we mean a sequent with* φ *in the antecedent and* ψ *in the succedent in the context of arbitrary sets of formulae* Δ *and* Γ, *i.e.,* $\Delta \cup \{\varphi\} \vdash \Gamma \cup \{\psi\}$.

A sequent $\Delta \vdash \Gamma$ *is* valid *iff*

$$\forall \mathcal{M} : Model \bullet \mathcal{M} \models \Delta \vdash \Gamma,$$

that is, it holds for all models.

In the following, we introduce a sequent calculus to prove properties over dCSP formulae. The calculus consists of *rule schemata* of the shape

$$\frac{\Phi_1 \vdash \Psi_1 \quad \cdots \quad \Phi_n \vdash \Psi_n}{\Phi \vdash \Psi}$$

that can be instantiated with arbitrary contexts Δ, Γ to rules of the form

$$\frac{\Delta, \Phi_1 \vdash \Psi_1, \Gamma \quad \cdots \quad \Delta, \Phi_n \vdash \Psi_n, \Gamma}{\Delta, \Phi \vdash \Psi, \Gamma}.$$

If the rule schema contains no sequent symbol, e.g.,

$$\frac{\Phi}{\Psi},$$

then it defines rules that can be used symmetrically to both sides of a sequent in arbitrary contexts, so it represents

$$\frac{\Delta, \Phi \vdash \Gamma}{\Delta, \Psi \vdash \Gamma} \quad \text{and} \quad \frac{\Delta \vdash \Phi, \Gamma}{\Delta \vdash \Psi, \Gamma}.$$

As usual, formulae above the line are premises and the formula below the line the consequence: if the premises (and possibly some side-conditions) are true, then the consequence also holds. The basic idea, that the proof rules have in common, is to syntactically reduce a proof goal to smaller formulae that have less operators or contain a dCSP formula with smaller processes. However, the rules are usually applied from bottom to top, from the conclusion to the premises. That is, starting from a proof goal the rules are applied backwards—by this, the sequent formulae are stepwise reduced—until every branch of the resulting proof tree is closed by an axiom.

Definition 4.2.2 (Derivability)
A formula ψ is derivable *in the sequent calculus from a formula set Φ_0, written $\Phi_0 \vdash_{SC} \psi$, if there is a proof tree that proves $\Phi_0 \vdash \psi$, that is, a tree of sequents with the following properties:*

- *$\Phi_0 \vdash \psi$ is the root*
- *every leaf is an axiom from the calculus*
- *for every node $\Delta, \Phi \vdash \Psi, \Gamma$ with children $\Delta, \Phi_i \vdash \Psi_i, \Gamma$ there is a rule schema*

$$\frac{\Phi_1 \vdash \Psi_1 \quad \cdots \quad \Phi_n \vdash \Psi_n}{\Phi \vdash \Psi}$$

(or a corresponding rule) in the calculus.

Of course, we are only interested in sound proofs. The sequent calculus is *sound* iff $\Delta \vdash_{SC} \Gamma$ always implies that $\Delta \vdash \Gamma$ is valid, which is the case if all proof rules are sound. Hence, we define soundness and the stronger notion of local soundness.

Definition 4.2.3 (Soundness)
A rule schema

$$\frac{\Phi_1 \vdash \Psi_1 \quad \cdots \quad \Phi_n \vdash \Psi_n}{\Phi \vdash \Psi}$$

is called locally sound *iff the conclusion follows from the premises in arbitrary contexts Δ, Γ (with fixed model):*

$$\forall \mathcal{M} : \mathit{Model} \bullet \big(\mathcal{M} \models (\Delta, \Phi_1 \vdash \Psi_1, \Gamma) \text{ and } \ldots \text{ and } \mathcal{M} \models (\Delta, \Phi_n \vdash \Psi_n, \Gamma) \\ \text{implies} \\ \mathcal{M} \models (\Delta, \Phi \vdash \Psi, \Gamma)\big)$$

It is called sound *iff the validity of the conclusion follows from the validity of the premises:*

$$\begin{gathered}\forall \mathcal{M} : \mathit{Model} \bullet \mathcal{M} \models (\Delta, \Phi_1 \vdash \Psi_1, \Gamma) \text{ and} \\ \vdots \\ \forall \mathcal{M} : \mathit{Model} \bullet \mathcal{M} \models (\Delta, \Phi_n \vdash \Psi_n, \Gamma) \\ \textit{implies} \\ \forall \mathcal{M} : \mathit{Model} \bullet \mathcal{M} \models (\Delta, \Phi \vdash \Psi, \Gamma)\end{gathered}$$

Note that local soundness implies soundness.

4.3. Proof Rules

In the following sections, we introduce the rule schemata establishing our sequent calculus. First, we include standard rules to reason about propositional and first-order formulae. Afterwards, we enrich this set of rules by rules for symbolic execution of dCSP processes. The appendix contains a list of all sequent rules on page 279.

4.3.1. Structural Rules

The structural rules are needed for basic modifications: unnecessary formulae may be dropped from a sequent, or we may split up a proof goal into two sub-goals using an auxiliary formula and the cut rule. Since we consider sequences to be sets of formulae we do not need rules for contraction and permutation of sequent elements. It is a conclusion of Gentzen's Hauptsatz [Gen35] that the cut rule is actually not necessary, but it is included to simplify proofs.

$$\frac{\vdash}{\gamma \vdash} \qquad \text{(weakening left)}$$

$$\frac{\vdash}{\vdash \gamma} \qquad \text{(weakening right)}$$

$$\frac{\gamma \vdash \qquad \vdash \gamma}{\vdash} \qquad \text{(cut)}$$

4.3.2. Propositional Rules

There are two rules for all propositional operators and an axiom rule stating that φ always implies φ.

$$\frac{}{\varphi \vdash \varphi} \qquad \text{(axiom)}$$

$$\frac{\vdash \varphi}{\neg\varphi \vdash} \quad \text{(negation left)}$$

$$\frac{\varphi \vdash}{\vdash \neg\varphi} \quad \text{(negation right)}$$

$$\frac{\varphi, \psi \vdash}{\varphi \wedge \psi \vdash} \quad \text{(and left)}$$

$$\frac{\vdash \varphi \quad \vdash \psi}{\vdash \varphi \wedge \psi} \quad \text{(and right)}$$

$$\frac{\varphi \vdash \quad \psi \vdash}{\varphi \vee \psi \vdash} \quad \text{(or left)}$$

$$\frac{\vdash \varphi, \psi}{\vdash \varphi \vee \psi} \quad \text{(or right)}$$

$$\frac{\psi \vdash \quad \vdash \varphi}{\varphi \Rightarrow \psi \vdash} \quad \text{(implication left)}$$

$$\frac{\varphi \vdash \psi}{\vdash \varphi \Rightarrow \psi} \quad \text{(implication right)}$$

4.3.3. First-Order Rules

For quantified formulae, we also use the standard sequent calculus proof rules.

$$\frac{\varphi[t/x], \forall\, x : T \bullet \varphi \vdash}{\forall\, x : T \bullet \varphi \vdash} \quad \text{(all left)}$$

$$\frac{\vdash \varphi[y/x]}{\vdash \forall\, x : T \bullet \varphi} \quad \text{(all right)}$$

$$\frac{\varphi[y/x] \vdash}{\exists\, x : T \bullet \varphi \vdash} \quad \text{(exists left)}$$

$$\frac{\vdash \varphi[t/x], \exists\, x : T \bullet \varphi}{\vdash \exists\, x : T \bullet \varphi} \quad \text{(exists right)}$$

The term t is of type T, i.e., $t \in T$, and y is a fresh variable of type T not occurring elsewhere.

4.3.4. Symbolic Execution of dCSP Formulae

In the following, we introduce new proof rules to handle dCSP formulae that extend the standard proof rules we have presented above. They execute the CSP programs with data constraints occurring in the formulae symbolically, and unwind the programs along their process structure.

We use the following conventions in the definition of the rules: φ always represents a predicate, δ a dCSP formula, and γ a path formula of the form $\Box\delta$, $\Diamond\delta$, or a dCSP formula. In the proof rules, we write for the sake of conciseness $a \to P$ instead of $a \bullet \varphi \to P$ if the constraint φ is of no relevance for the rule. In addition, we abbreviate formulae $\langle\!\![a \to \texttt{Skip}]\!\!\rangle\gamma$ by $\langle\!\![a]\!\!\rangle\gamma$. We recall that we use the combined operator $\langle\!\![\cdot]\!\!\rangle$ if a rule applies to $[\,\cdot\,]$ and $\langle\,\cdot\,\rangle$. The process Q in the first rule is defined by $Q \stackrel{c}{=} P$.

$$\frac{\langle\!\![P]\!\!\rangle\gamma}{\langle\!\![Q]\!\!\rangle\gamma} \qquad \text{(process call)}$$

$$\frac{\delta}{\langle\!\![\texttt{Skip}]\!\!\rangle\delta} \qquad (\text{skip}_\delta)$$

$$\frac{\varphi}{[\texttt{Skip}]\Box\varphi} \qquad (\text{skip}_\Box)$$

$$\frac{\varphi}{\langle\texttt{Skip}\rangle\Diamond\varphi} \qquad (\text{skip}_\Diamond)$$

$$\frac{\langle\!\![a]\!\!\rangle\langle\!\![P]\!\!\rangle\delta}{\langle\!\![a \to P]\!\!\rangle\delta} \qquad (\text{prefix}_\delta)$$

$$\frac{[a]\Box\varphi \wedge [a][P]\Box\varphi}{[a \to P]\Box\varphi} \qquad (\text{prefix}_\Box)$$

$$\frac{\langle a\rangle\Diamond\varphi \vee \langle a\rangle\langle P\rangle\Diamond\varphi}{\langle a \to P\rangle\Diamond\varphi} \qquad (\text{prefix}_\Diamond)$$

$$\frac{[P_1]\gamma \wedge [P_2]\gamma}{[P_1 \,\Box\, P_2]\gamma} \qquad \text{(box choice)}$$

$$\frac{\langle P_1\rangle\gamma \vee \langle P_2\rangle\gamma}{\langle P_1 \,\Box\, P_2\rangle\gamma} \qquad \text{(diamond choice)}$$

$$\frac{\langle\!\![P_1]\!\!\rangle\langle\!\![P_2]\!\!\rangle\delta}{\langle\!\![P_1 \,{}_{9}^{\circ}\, P_2]\!\!\rangle\delta} \qquad (\text{sequence}_\delta)$$

$$\frac{([P_1]\Box\varphi) \wedge ([P_1][P_2]\Box\varphi)}{[P_1 \,{}_{9}^{\circ}\, P_2]\Box\varphi} \qquad (\text{sequence}_\Box)$$

$$\frac{(\langle P_1\rangle\Diamond\varphi) \vee (\langle P_1\rangle\langle P_2\rangle\Diamond\varphi)}{\langle P_1 \,{}_{9}^{\circ}\, P_2\rangle\Diamond\varphi} \qquad (\text{sequence}_\Diamond)$$

These rules correspond to the operators of the process expressions as defined in Sect. 3.1.1. None of them contains a sequent symbol, so the rules can all be applied on both sides of a sequent. The rule (process call) is used to replace a process identifier with its defining process in a dCSP expression. Generally, there are four rules for every operator, because we need different rules to cover the temporal case and the non-temporal, as well as the box and the diamond case, in all combinations. For the non-temporal case, the box and the diamond rules are symmetric. Thus, we summarise both in one rule using the $\llparenthesis \cdot \rrparenthesis$ notation, cf. (skip$_\delta$), (prefix$_\delta$), and (sequence$_\delta$). In the following, we denote the non-temporal case with δ-case and the temporal case with $\Box$- or $\Diamond$-case.

For the Skip process, the rules reflect the fact that Skip does not change any variables. The rules replace the Skip expression with the property we want to check—a property holds during or after a Skip if it is already valid without the Skip.

In the δ-case of the prefix operator, we want to check that after all executions (for the box operator) of $a \to P$ the property δ is true. The rule (prefix$_\delta$) says that we can instead verify that after all executions of a the dCSP formulae $[P]\delta$ is valid. So, the rule splits the process $a \to P$ into two parts, which can be handled with further rules in the calculus. The diamond case is treated in the same way. For the box version of the rule (prefix$_\Box$) we need to check that a formula φ is valid everywhere on all executions of $a \to P$. To this end, we need to prove that $\Box\varphi$ holds for every execution of a—or more precisely for every execution of the process $a \to$ Skip—and we need to prove that $[P]\Box\varphi$ holds *after* every execution of a. So, the rule reflects exactly this conjunction. As we will see later (cf. the description to rule (box step)), the former formula $[a]\Box\varphi$ also requires that a holds at the beginning of every execution.

The idea behind the remaining rules up to (sequence$_\Diamond$) is very similar. The rule (box choice) splits up a choice of processes into a conjunction of formulae. The diamond rules (prefix$_\Diamond$) and (diamond choice) are equal to their box versions except that a disjunction is used instead of a conjunction, because we do not need to check every sub-process, but only that the formula holds for one possible sub-process. The sequence rules are built-up identical to the prefix rules, which is not surprising as a prefix process $a \to P$ can be transformed into $a \to$ Skip $\fatsemi P$. So, we could omit the prefix rules and always replace a prefix operation by such a sequential composition, but it is more convenient to have rules for directly handling prefix operations.

The following step rules actually perform the *execution* of an event step with a corresponding data constraint. In doing so, the constrained event is consumed and replaced by a new formula representing the data change according to the event's constraint. A formula $\psi_{\overline{v}}^{\overline{v}_0}$ denotes the replacement of variables $\overline{v}$ with variables $\overline{v}_0$ for all unprimed variables v in ψ. Analogously, $\psi_{\overline{v}'}^{\overline{v}_0}$ denotes the replacement of all primed variables $\overline{v}'$ in ψ by $\overline{v}_0$.

$$\frac{\forall\, \overline{v}_0 \bullet \psi_{\overline{v}'}^{\overline{v}_0} \Rightarrow \delta_{\overline{v}}^{\overline{v}_0}}{[a \bullet \psi]\delta} \qquad \text{(box step}_\delta\text{)}$$

$$\frac{\exists\, \overline{v}_0 \bullet (\psi^{\overline{v}_0}_{\overline{v}'} \wedge \delta^{\overline{v}_0}_{\overline{v}})}{\langle a \bullet \psi \rangle \delta} \qquad \text{(diamond step}_\delta\text{)}$$

$$\frac{\varphi \wedge [a \bullet \psi]\varphi}{[a \bullet \psi]\Box\varphi} \qquad \text{(box step)}$$

$$\frac{\varphi \vee \langle a \bullet \psi \rangle \varphi}{\langle a \bullet \psi \rangle \Diamond \varphi} \qquad \text{(diamond step)}$$

The rule (box step$_\delta$) replaces the dCSP expression $[a \bullet \psi \rightarrow \texttt{Skip}]\delta$ by an implication that does not contain the event a. Events are only required for synchronisation of CSP processes and in the sequential situation, where this rule can be applied, the event a is not necessary anymore. The constraint ψ of the event a generally contains primed and unprimed variables, where the former relates to the post-state of the operation and the latter to the pre-state. With the replacement $\psi^{\overline{v}_0}_{\overline{v}}$ we introduce new variables $\overline{v}_0$ for the post-state of ψ, by which the post-state can be accessed in the δ-part of the dCSP formula. The δ is a dCSP formula that needs to be valid for all executions of $a \bullet \psi$ and that may contain further dCSP expressions. In this, we replace all pre-state variables occurring in ψ with the newly introduced variables. Thus, in the implication $\psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}}$ the post-state variables of the left side and the corresponding pre-state variables of the right side coincide. Hence, the state change of the ψ constraint is symbolically executed. Note that in the replacement $\delta^{\overline{v}_0}_{\overline{v}}$ actually *all* occurrences of $\overline{v}$ are replaced even if they occur in a constraint below a diamond or a box operator in sub-expressions of δ.

In rule (diamond step$_\delta$) the idea of of symbolically executing the state change as specified in the constraint ψ is similar to rule (box step$_\delta$). The difference is that we need to check that there actually is a possible execution. Hence, the rule demands that there exists $\overline{v}_0$ so that $\psi^{\overline{v}_0}_{\overline{v}'}$ and $\delta^{\overline{v}_0}_{\overline{v}}$ are valid.

The rules (box step) and (diamond step) do not consume the event a from the process expression. Instead, to prove that a formula φ holds always for all interpretations of a process $a \bullet \psi$, i.e., φ holds before and after all executions of a, the rule reduces this proof task to checking that φ is valid in the current context, and it is also valid after the execution of a. In the diamond case, the rule ensures that φ is valid either in the current context or after execution of a.

Finally, we also define rules to handle parallel composition of processes. In the following rule (parallel$_\Box$), P and Q are in guarded normal form. That is,

$$P = (\Box_{i \in 1..n}\ a_i \bullet \varphi_i \rightarrow P_i) \,\Box\, (\Box_{i \in 1..\overline{n}}\ b_i \bullet \varphi_{n+i} \rightarrow P_{n+i})$$

$$Q = (\Box_{i \in 1..m}\ \overline{a}_i \bullet \psi_i \rightarrow Q_i) \,\Box\, (\Box_{i \in 1..\overline{m}}\ c_i \bullet \psi_{m+i} \rightarrow Q_{m+i})$$

with $n_0 \leq min\{n, m\}$, and $a_i = \overline{a}_i$ for all $i \leq n_0$, and $a_i \neq \overline{a}_i$ for $i > n_0$. Furthermore, let $a_j \in A$, $b_j, c_j \notin A$ for all j.

$$\frac{\begin{array}{l} \quad [a_1 \bullet (\varphi_1 \wedge \psi_1) \rightarrow (P_1 \parallel_A Q_1)]\gamma \\ \wedge \ \ldots \\ \wedge \ [a_{n_0} \bullet (\varphi_{n_0} \wedge \psi_{n_0}) \rightarrow (P_{n_0} \parallel_A Q_{n_0})]\gamma \\ \wedge \ [b_1 \bullet \varphi_{n+1} \rightarrow (P_{n+1} \parallel_A Q)]\gamma \\ \wedge \ \ldots \\ \wedge \ [c_{\overline{m}} \bullet \psi_{m+\overline{m}} \rightarrow (P \parallel_A Q_{m+\overline{m}})]\gamma \end{array}}{[P \parallel_A Q]\gamma} \quad (\text{parallel}_\Box)$$

$$\frac{\langle a_i \rightarrow (P_i \parallel_A Q)\rangle\gamma}{\langle P \parallel_A Q\rangle\gamma} \quad (\text{interleaving}_\Diamond)$$

$$\frac{\langle a_j \rightarrow (P_j \parallel_A Q_k)\rangle\gamma}{\langle P \parallel_A Q\rangle\gamma} \quad (\text{sync}_\Diamond)$$

The rule (parallel$_\Box$) is a derived rule, because the parallel process in the conclusion of the rule is equivalent (with respect to the interpretation semantics of eCSP) to the choice of the processes from the premises. Thus, with the application of rule (process equivalence) and rule (box choice) we get the soundness of rule (parallel$_\Box$).

In rules (interleaving$_\Diamond$) and (sync$_\Diamond$), P and Q are in guarded normal form with $P = a_1 \rightarrow P_1 \Box \cdots \Box a_n \rightarrow P_n$ and $Q = b_1 \rightarrow Q_1 \Box \cdots \Box b_n \rightarrow Q_m$. Let $a_i \notin A$ for rule (interleaving$_\Diamond$) and $a_j = b_j \in A$ for rule (sync$_\Diamond$). For the diamond case, we only need to show that there is at least one execution of the parallel composition, such that the desired formula γ is valid. That is, we only need to find an appropriate unwinding of $P \parallel_A Q$. So, rule (interleaving$_\Diamond$) proves that there is such an unwinding for executions starting without a synchronisation, whereas rule (sync$_\Diamond$) unwinds executions starting with a synchronisation.

We make use of the auxiliary rule (process equivalence) to transform CSP processes into guarded normal form or to apply the rules to the symmetric cases, for instance to unwind process Q instead of P in rule (interleaving$_\Diamond$).

4.3.5. Symbolic Execution of dCSP Specifications with Unknown Processes

We need rules to handle unknown processes with temporal constraints. The idea is not to treat these constraints in our calculus directly. Instead, we call an external procedure that checks if the constraints of the unknown process actually ensure properties by which the remaining proof can be completed. Thus, the rules we give here for unknown processes can be seen as oracle rules that access external techniques to reason over the temporal constraints. Hence, the rules directly reflect the semantics

of constrained unknown processes. By this means, arbitrary timed logics to formulate assumptions on unknown processes are integrated in our approach.

Rule (assumption axiom) is an axiom expressing the trivial fact that after termination of all non-terminating processes everything is true.

$$\frac{}{\psi \vdash [\mathtt{Proc}^{\infty}_{\backslash A,V} \bullet F]\delta} \quad \text{(assumption axiom)}$$

For the remaining rules, the interesting part is contained in the side-conditions. Rule (box assumption$_\delta$) is sound if for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ and all interpretations $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ with terminating model $\overline{\mathcal{M}}$ the formula φ holds: $\overline{\mathcal{M}} \models \varphi$. This describes exactly the interpretations of $[\mathtt{Proc}_{\backslash A,V} \bullet F]\varphi$ for models with $\mathcal{M} \models \psi$. Further, the conclusion of the rule demands that φ implies δ. The rule only applies to terminating unknown processes; for the infinite case the unknown process cannot be consumed. Note that this rule contains sets of formulae Δ and Γ and, thus, it is actually a rule and not a rule schema according to the definition in Sect. 4.2. In particular, the premise of rule (box assumption$_\delta$) does actually not contain the context formulae of the conclusion. So, the premise proves that δ follows from φ, independent from the specific contexts Δ and Γ.

$$\frac{\varphi \vdash \delta}{\Delta, \psi \vdash [\mathtt{Proc}_{\backslash A,V} \bullet F]\delta, \Gamma} \quad \text{(box assumption}_\delta\text{)}$$

Analogously, there is rule (diamond assumption$_\delta$) to handle constrained unknowns for $\langle\cdot\rangle$. The side-condition is here that for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ there is an interpretation $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ with terminating model $\overline{\mathcal{M}}$ for which the formula φ holds: $\overline{\mathcal{M}} \models \varphi$. The rule again applies only to terminating processes.

$$\frac{\varphi \vdash \delta}{\Delta, \psi \vdash \langle\mathtt{Proc}_{\backslash A,V} \bullet F\rangle\delta, \Gamma} \quad \text{(diamond assumption}_\delta\text{)}$$

Remark 4.3.1. It is not obvious that a formula like $\langle\mathtt{Proc}_{\backslash A,V} \bullet F\rangle\delta$ actually makes sense, because the process may loop infinitely and decide to never terminate. Anyway, the semantics of $\langle\cdot\rangle\delta$ only demands that there is an interpretation in the semantics with a terminating model that ensures δ. For an unconstrained process $\mathtt{Proc}_{\backslash A,V}$, this is always the case – for a constrained process $\mathtt{Proc}_{\backslash A,V} \bullet F$ it only depends on the constraint. Whether an unknown process `Proc` eventually terminates or not is a fairness question, which we do not examine here.

In the same way, we define the proof rules for the $\Box$-case: the side condition that must be proven for application of rule (assumption$_\Box$) is that for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ and all interpretations $\mathcal{I} \in [\![\mathtt{Proc}^{(\infty)}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ the formula $\Box\varphi$ holds: $\mathcal{I} \models \Box\varphi$. We recall that with $\mathtt{Proc}^{(\infty)}$ we express that the rule scheme may apply to both to `Proc` and to $\mathtt{Proc}^{\infty}$.

$$\frac{}{\psi \vdash [\mathtt{Proc}^{(\infty)}_{\backslash A,V} \bullet F]\Box\varphi} \quad \text{(assumption}_\Box\text{)}$$

With the side-condition that for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ there is an interpretation $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ for that the formula $\Diamond\varphi$ holds, i.e., $\mathcal{I} \models \Diamond\varphi$, the rule (assumption$_\Diamond$) is defined by:

$$\frac{}{\psi \vdash \langle \mathtt{Proc}^{(\infty)}_{\backslash A,V} \bullet F \rangle \Diamond\varphi} \quad (\text{assumption}_\Diamond)$$

In this way, we can use an arbitrary proof method for the temporal logic if it is possible to check the side-conditions of rules (box assumption$_\delta$) up to (assumption$_\Diamond$) in this logic. In this work, we typically use the Duration Calculus (DC) [ZHR91] as timed logic for unknown processes. Thus, we apply the approach of [MFHR08] to solve the constraints via model checking, but we could also use alternative rule-based approaches like a calculus for DC [SS94].

We do not have rules for parallel composition of unknown processes here. Parallelism over unknown processes causes some interference problems. Therefore, we dedicate a section to this problem and discuss possible solutions in Sect. 6.1.

4.3.6. Induction Rules

In contrast to standard Dynamic Logic over while programs, dCSP expresses properties over recursive processes. Hence, we provide some induction rules to allow reduction of recursion in CSP expressions. These rules are variants of the *Proof Rule for Recursion* and the *Fixed point Induction Rule* of Roscoe [Ros98], but they are adapted to our needs in the context of the Dynamic Logic extension. In the following rule, we consider a simple recursive process of the shape $P \equiv Q \fatsemi P$, in which Q does not contain any further references to P. Note that the rule is actually a rule and not a rule schema. In particular, in the middle and in the right formula the context formulae Δ and Γ are not considered.

$$\frac{\Delta \vdash \varphi_{in}, \Gamma \qquad \varphi_{in} \vdash [Q]\gamma \qquad \varphi_{in} \vdash [Q]\varphi_{in}}{\Delta \vdash [P]\gamma, \Gamma} \quad (\text{box loop})$$

$$\frac{\Delta \vdash \varphi_{in}(\overline{y}), \Gamma \qquad \varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [Q^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [F(Q)]\gamma(\overline{y})}{\Delta \vdash [P]\gamma(\overline{y}), \Gamma} \quad (\text{box loop gen})$$

In both rules, the formula γ may represent the temporal and the non-temporal case, so γ can be a formula $\Box\varphi$ or δ (but not $\Diamond\varphi$). In (box loop gen), P is a recursive process expression defined by $P \overset{c}{=} F(P)$. The formula $\gamma(\overline{y})$ is a formula over a vector of all function symbols occurring in the sequent. The formula $\varphi_{in}(\overline{y})$ is an initial condition that is true at every beginning of the recursion. The formulae $\gamma(\overline{y})$ and $\varphi_{in}(\overline{y})$ do not contain free function symbols besides $\overline{y}$. The rule (box loop gen) is a generalisation of the simpler rule (box loop), but it is sufficient to apply (box loop) in many cases. Moreover, in (box loop) the ideas of both fixed point induction rules become clear: if we want to prove that a formula γ holds always on every execution

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{(axiom)}}{x_3 \leq 8 \vdash x_3 \leq 8, [Q_x^{x_3}]\Box x_3 \leq 10} \quad \dfrac{\text{(axiom)}}{x_3 \leq 8, [Q_x^{x_3}]\Box x_3 \leq 10 \vdash [Q_x^{x_3}]\Box x_3 \leq 10}}{x_3 \leq 8, (x_3 \leq 8 \Rightarrow [Q_x^{x_3}]\Box x_3 \leq 10) \vdash [Q_x^{x_3}]\Box x_3 \leq 10}\text{(implication left)}}{x_3 \leq 8, H \vdash [Q_x^{x_3}]\Box x_3 \leq 10}\text{(all left)}}{x \leq 8, x_1 = x+1, x_2 = x_1+1, x_3 = x_2 - 2, H \vdash [Q_x^{x_3}]\Box x_3 \leq 10}\text{(arith)}}{\vdots}\text{(implication right)}$$

$$\dfrac{\dfrac{\dfrac{\text{(arith)}}{x = 0 \vdash x \leq 8} \quad \dfrac{\dfrac{\dfrac{(*)}{x \leq 8, H \vdash [a]\Box x \leq 10} \quad \dfrac{\dfrac{\dfrac{\vdots}{x \leq 8, x_1 = x+1, H \vdash [a_1 \to b_1 \to Q_x^{x_1}]\Box x_1 \leq 10}\text{(implication right)}}{x \leq 8, H \vdash x_1 = x+1 \Rightarrow [a_1 \to b_1 \to Q_x^{x_1}]\Box x_1 \leq 10}}{x \leq 8, H \vdash [a][a \to b \to Q]\Box x \leq 10}\text{(box step}_\delta\text{)}}{x \leq 8, H \vdash [a]\Box x \leq 10 \wedge [a][a \to b \to Q]\Box x \leq 10}\text{(and right)}}{x \leq 8, H \vdash [F(Q)]\Box x \leq 10}\text{(prefix}_\Box\text{)}}{x = 0 \vdash [F(P)]\Box x \leq 10}\text{(box loop gen)}$$

Figure 4.1.: Proof tree for Example 4.3.2

of P, that consists of an infinitely often repeated sub-process Q, then we first show that γ holds for Q under the assumption that a starting condition φ_{in} is valid. In addition, we need to show that after all executions of Q the same starting condition is valid as for the first execution of Q, by which γ remains valid for the next execution of Q and so on. Thus, the rule requires to prove $\Delta \vdash \varphi_{in}, \Gamma$ stating that the starting condition φ_{in} is valid in the current context. And it requires to show that $[Q]\gamma$ is a consequence from the starting condition φ_{in} and so is $[Q]\varphi_{in}$, which corresponds to the intuition that after execution of Q the same starting condition holds again.

The rule (box loop gen) generalises this idea to arbitrary recursive processes. We again need to show that an initial condition $\varphi_{in}(\overline{y})$ holds in the current context. The rule bases on an inductive argument: assuming that φ_{in} implies that γ holds for an arbitrary process Q, we must show that γ also holds for $F(Q)$. The induction hypothesis is given by

$$\forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [Q_{\overline{y}}^{\overline{x}}]\gamma(\overline{x})).$$

It states that regardless of how φ_{in} is instantiated with function symbols $\overline{x}$, if φ_{in} is valid for these $\overline{x}$, then $[Q_{\overline{y}}^{\overline{x}}]\gamma(\overline{x})$ is also valid. In this term all function symbols $\overline{y}$ in Q need to be replaced by $\overline{x}$, because of the function symbol replacements performed when symbolically executing the process $F(Q)$: the induction hypothesis can only be applied if its function symbols correspond to the function symbols introduced in the symbolic execution.

Example 4.3.2. To further illustrate this, we prove

$$x = 0 \vdash [F(P)]\Box x \leq 10,$$

where $F(P)$ is the recursive process

$$F(P) \overset{c}{=} a \bullet (x' = x + 1) \to a \bullet (x' = x + 1) \to b \bullet (x' = x - 2) \to P.$$

With a small equivalence conversion, we could use rule (box loop) for the proof, but for demonstration we apply the rule (box loop gen) with $\varphi_{in}(x) \leq 8$ followed by three applications of (box step$_\delta$). We abbreviate the induction hypothesis by

$$H := \forall\, z \bullet (z \leq 8 \Rightarrow [Q_x^z]\Box z \leq 10).$$

The proof tree for the desired property is pictured in Fig. 4.1. We omit the constraints of the events for the sake of readability. The goal (∗) is easy to close using (among structural and arithmetic rules) the rules (box step) and (box step$_\delta$); the induction hypothesis is not needed here. In the dotted part of the proof tree, the previous three proof steps are repeated two times to reduce the remaining process $a \rightarrow b \rightarrow Q_x^{x_1}$. The interesting part of the proof tree is the application of (box loop gen) in the first (bottom-line) proof step and the instantiation of the induction hypothesis with rule (all left). There, we instantiate the variable z with the current function symbol of the formula we want to prove, which is x_3. Hence, in the last proof step, we resolve the implication of the hypothesis and need to prove that the initial condition $x_3 \leq 8$ is valid (left branch) by which we directly get the desired validity of formula $[Q_3]\Box x_3 \leq 10$ (right branch) and our proof is finished.

Currently, we have no induction rules for the diamond cases. Diamond properties are proven by unwinding processes as long as the desired property is not true. The proof finishes if one can show that a property holds after n unwindings. This is possible with the rules from Sect. 4.3.4 but it only works if the number of necessary unwindings is fixed. Otherwise, we need an induction rule again that must comprise a termination argument (for the δ-case of the rule). As our focus is to prove safety properties we do not examine such termination checking proof rules here. Proof rules for checking termination can be found, e.g., in [AdBO09].

4.3.7. Auxiliary dCSP Rules

At last, we also define some simple auxiliary rules that can be used to transform processes and that may help when performing proofs with our calculus.

If P and Q are equivalent CSP processes: $P \equiv Q$ we can apply the following rules to replace P with Q.

$$\frac{\langle\!| Q |\!\rangle \gamma}{\langle\!| P |\!\rangle \gamma} \qquad \text{(process equivalence)}$$

$$\frac{[P]\Box\varphi}{[P]\varphi} \qquad (\Box\text{-introduction})$$

$$\frac{\delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0}}{[Q]\delta \vdash [Q]\epsilon} \qquad \text{(process step)}$$

Rule (□-introduction) says that, if we know that on all executions of P always φ holds, then we can conclude that after all executions φ holds, because the latter is

simply a special case of the former. This rule can be very helpful in combination with rule (box loop), because when applying this rule we often have to reduce two proof branches which do only differ in the $\Box$-operator. Thus, with rule ($\Box$-introduction) we can use one proof branch where we proved $[P]\Box\varphi$ to close the other with $[P]\varphi$. Rule (process step) allows us to consume a process if it occurs on both sides of the sequent symbol. Since the process Q possibly changes system variables we have to replace all system variables occurring in the conclusion by fresh symbols v_0. This rule is for instance helpful to prove the soundness of rule (box loop).

Remark 4.3.3 (Rules for Data Structures). To reason about data structures, we follow the approach of, e.g., [Rei95]: arbitrary theories can be integrated into our logic by specifying a set of axioms that describe an algebraic data type over an appropriate signature, for instance natural numbers or lists. These axioms are used as implicit premises in sequent-style proofs. For practical applications it is convenient to derive corresponding proof rules dedicated to commonly used theories. In our examples, we use the theory of real numbers and additional rules to reason over relations on real numbers. For instance, in the proof tree of Example 4.3.2 in Fig. 4.1, we use an implicit rule (arith) to indicate arithmetical conversions in the theory of real numbers. In this thesis, we focus on the general case and do not explicitly introduce rules for specific theories (cf. [Rei95, BHS07]).

4.4. Soundness of the Calculus

We examine the soundness of our calculus and discuss its incompleteness in the following theorems.

Theorem 4.4.1 (Soundness)
The calculus as presented in Sect. 4.3 is sound, i.e., validity follows from derivability in the calculus.

Proof. To prove the soundness of the calculus, we need to prove soundness of every single rule from Sect. 4.3. For the standard first-order rules (weakening left) up to (exists right), the proof is as usual. Thus, we only give the soundness proof for the dCSP rules here. First, we show the soundness of the rules for the box case and then the rules for the diamond case.
Box cases:

Rule (process equivalence): We prove this rule in the beginning, because it is used in some of the proofs for the remaining rules. The side-condition $P \equiv Q$ implies that every interpretation of P is also an interpretation of Q and vice versa. Let $\mathcal{M} \models [Q]\gamma$, i.e., for every interpretation in $[\![Q]\!]\mathcal{M}$ the formula γ is valid. Due to equivalence of P and Q for an arbitrary interpretation $\mathcal{I} \in [\![P]\!]\mathcal{M}$, $\mathcal{I} \in [\![Q]\!]\mathcal{M}$ is true. Due to $\mathcal{I} \models \gamma$[1] we get the desired $\mathcal{M} \models [P]\gamma$.

[1]We recall that γ represents a temporal or a non-temporal formula, i.e., $\mathcal{I} \models \gamma$ means (1) in the

Rule (process call): We utilise the fact that $Q \stackrel{c}{=} P$ implies the equivalence $Q \equiv P$ in trace and interpretation semantics. By this, we get the soundness of rule (process call) by applying rule (process equivalence).

Rule (skip$_\delta$): We prove that

$$\frac{\delta}{[\texttt{Skip}]\delta}$$

is locally sound. So, let $\mathcal{M} \models \delta$. We need to show that at the end of every interpretation $\mathcal{I} \in [\![\texttt{Skip}]\!]\mathcal{M}$ the desired property δ is valid. This is true, because $[\![\texttt{Skip}]\!]\mathcal{M}$ contains exactly one interpretation of the shape $\langle \mathcal{M}, \checkmark, \mathcal{M} \rangle$, i.e., $\mathcal{M}$ is the terminating model with $\mathcal{M} \models \delta$. Thus, $\mathcal{M} \models [\texttt{Skip}]\delta$.

Since the rule can be applied to both sides of the sequent symbol, we also need to show the other direction, i.e., $\mathcal{M} \models \neg\delta$ implies $\mathcal{M} \models \neg[\texttt{Skip}]\delta$, which is equivalent to $\mathcal{M} \models [\texttt{Skip}]\delta$ implies $\mathcal{M} \models \delta$. The argument is here, and in the following rules, completely analogue to the first implication.

Rule (skip$_\Box$): This rule is also locally sound: with the same argumentation as in the previous case the interpretation $\mathcal{I}$ satisfies $\Box\varphi$, i.e., $\mathcal{M} \models [\texttt{Skip}]\Box\varphi$.

Rule (box choice): Let $\mathcal{M} \models [P_1]\gamma \wedge [P_2]\gamma$, which means that $\mathcal{M} \models [P_1]\gamma$ and $\mathcal{M} \models [P_2]\gamma$. We examine $\mathcal{I} \in [\![P_1 \Box P_2]\!]\mathcal{M}$ with terminating model $\mathcal{M}'$ and show that $\mathcal{M}' \models \gamma$. There are two cases: $\mathcal{I}$ corresponds to an interpretation of P_1 or to an interpretation of P_2. Assuming that the former is true, then $\mathcal{I} \in [\![P_1]\!]\mathcal{M}$ with terminating model $\mathcal{M}'$. Due to the premise $\mathcal{M}' \models \gamma$. The argumentation is the same for the second case. By this, $\mathcal{M} \models [P_1 \Box P_2]\gamma$.

Rule (sequence$_\delta$): Let $\mathcal{M} \models [P_1][P_2]\delta$, which means that for every interpretation in $[\![P_1]\!]\mathcal{M}$ with terminating model $\mathcal{M}'$, $\mathcal{M}' \models [P_2]\delta$ is true, i.e., for every terminating model $\mathcal{M}''$ of interpretations in $[\![P_2]\!]\mathcal{M}'$ the formula δ holds: $\mathcal{M}'' \models \delta$.

We examine an interpretation $\mathcal{I} \in [\![P_1 \fatsemi P_2]\!]\mathcal{M}$ with terminating model $\overline{\mathcal{M}}$. The proof is finished if we can show $\overline{\mathcal{M}} \models \delta$. Note that if P_1 does not terminate then the proposition is trivially true, because the interpretations of $P_1 \fatsemi P_2$ have no terminating model. The interpretation $\mathcal{I}$ has the shape $\mathcal{I} = \langle \mathcal{M}, a_1, \mathcal{M}_1, \ldots, \checkmark, \overline{\mathcal{M}} \rangle$. Due to the semantics of $P_1 \fatsemi P_2$, we can decompose $\mathcal{I}$ into two sub-interpretations

$$\begin{aligned} \mathcal{I}_1 &:= \langle \mathcal{M}, a_1, \mathcal{M}_1, \ldots, \mathcal{M}_k, \checkmark, \mathcal{M}_k \rangle \\ \mathcal{I}_2 &:= \langle \mathcal{M}_k, a_{k+1}, \mathcal{M}_{k+1}, \ldots, \overline{\mathcal{M}}, \checkmark, \overline{\mathcal{M}} \rangle, \end{aligned}$$

such that $\mathcal{I}_1$ is interpretation of P_1 (starting with model $\mathcal{M}$ and terminating model $\mathcal{M}_k$) and $\mathcal{I}_2$ interpretation of P_2 (starting with model $\mathcal{M}_k$ and terminating model $\overline{\mathcal{M}}$).

---

case of $\gamma = \delta$ that δ is true for the terminating model of $\mathcal{I}$ or (2) in the case of $\gamma = \Box\varphi$ that φ is true everywhere on $\mathcal{I}$.

We now apply the premises (by instantiating $\mathcal{M}'$ with $\mathcal{M}_k$ and $\mathcal{M}''$ with $\overline{\mathcal{M}}$): due to $\mathcal{I}_1 \in [\![P_1]\!]\mathcal{M}$ with terminating model $\mathcal{M}_k$, we can conclude $\mathcal{M}_k \models [P_2]\delta$. With $\mathcal{I}_2 \in [\![P_2]\!]\mathcal{M}_k$ and the second premise, we get that for the terminating model $\overline{\mathcal{M}}$ of $\mathcal{I}_2$ the desired $\overline{\mathcal{M}} \models \delta$ is true.

Rule (sequence$_\Box$): Let $\mathcal{M} \models [P_1]\Box\varphi$ and $\mathcal{M} \models [P_1][P_2]\Box\varphi$. We examine $\mathcal{I} \in [\![P_1 \fatsemi P_2]\!]\mathcal{M}$ and need to show that $\mathcal{I} \models \Box\varphi$. Like for rule (sequence$_\delta$), we split $\mathcal{I}$ into

$$\begin{aligned} \mathcal{I}_1 &:= \langle \mathcal{M}, a_1, \mathcal{M}_1, \ldots, \mathcal{M}_k, \checkmark, \mathcal{M}_k \rangle \\ \mathcal{I}_2 &:= \langle \mathcal{M}_k, a_{k+1}, \mathcal{M}_{k+1}, \ldots, \mathcal{M}_n, \checkmark, \mathcal{M}_n \rangle. \end{aligned}$$

From the premise $\mathcal{M} \models [P_1]\Box\varphi$ we can conclude that

$$\forall\, i \in 1..k \bullet \mathcal{M}_i \models \varphi \tag{4.1}$$

and from the second premise (since $\mathcal{M}_k$ is terminating model of P_1) $\mathcal{M}_k \models [P_2]\Box\varphi$. Since $\mathcal{I}_2 \in [\![P_2]\!]\mathcal{M}_k$, we conclude $\mathcal{I}_2 \models \Box\varphi$, i.e.,

$$\forall\, i \in k..n \bullet \mathcal{M}_i \models \varphi. \tag{4.2}$$

From (4.1) and (4.2), we get $\mathcal{I} \models \Box\varphi$.

Rule (prefix$_\delta$): This is a derived rule because of the process equivalence $a \to P \equiv a \to \texttt{Skip} \fatsemi P$. We can then deduce

$$\cfrac{\cfrac{[a][P]\delta}{[a \to \texttt{Skip} \fatsemi P]\delta}\ \text{(sequence}_\delta\text{)}}{[a \to P]\delta.}\ \text{(process equivalence)}$$

Rule (prefix$_\Box$): Analogously to the previous case, we derive

$$\cfrac{\cfrac{[a]\Box\varphi \wedge [a][P]\Box\varphi}{[a \to \texttt{Skip} \fatsemi P]\Box\varphi}\ \text{(sequence}_\Box\text{)}}{[a \to P]\Box\varphi.}\ \text{(process equivalence)}$$

Rule (box step$_\delta$): In this rule, vectors of variables are replaced in formulae like $\delta^{\overline{v}_0}_{\overline{v}}$ to symbolically execute the state change of an operation. To simplify the presentation, we show the soundness of the rule considering $\overline{v}$ as a single variable, but keep in mind that $\overline{v}$ actually represents a vector of symbols and $\delta^{\overline{v}_0}_{\overline{v}}$ a replacement of multiple symbols v by corresponding v_0.

This rule is not locally sound (see below), so we have to take care on the context Δ, Γ, which is implicit in the rule. We need to prove two cases: (1) Assuming $\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \Delta \vdash \forall\, \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}}, \Gamma$, we have to show

$$\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \Delta \vdash [a \bullet \psi]\delta, \Gamma \tag{4.3}$$

and (2) assuming $\forall \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \Delta, \forall \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}} \vdash \Gamma$, we have to show

$$\forall \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \Delta, [a \bullet \psi]\delta \vdash \Gamma \tag{4.4}$$

In both cases, if for a model $\mathcal{M} \models \Delta \vdash \Gamma$, then $\mathcal{M}$ is also model of the formulae in (4.3) and (4.4) (because the sequent formulae are actually disjunctions). Thus, we only have to examine models with $\mathcal{M} \not\models \Delta \vdash \Gamma$. We start with the first case and consider such an $\mathcal{M}$ with $\mathcal{M} \not\models \Delta \vdash \Gamma$. Due to the assumption of case (1) it generally holds for every fresh symbol $\overline{v}_0$ not occurring elsewhere

$$\forall \mathcal{M} : \mathit{Model} \text{ if } \mathcal{M} \not\models \Delta \vdash \Gamma \text{ then } \mathcal{M} \models \psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}}. \tag{4.5}$$

We prove that $\mathcal{M} \models [a \bullet \psi]\delta$. Thus, we consider $\mathcal{I} \in [\![a \bullet \psi]\!]\mathcal{M}$ and show $\overline{\mathcal{M}} \models \delta$ for the terminating model $\overline{\mathcal{M}}$. All such interpretations have the shape

$$\mathcal{I} = \langle \mathcal{M}, (a \bullet \psi), \overline{\mathcal{M}}, \checkmark\overline{\mathcal{M}} \rangle.$$

If there is no suchlike interpretation, then the desired property is trivially true. The interpretation $\mathcal{I}$ implies, according the semantics of $a \bullet \psi$,

$$\mathcal{M} \cup \overline{\mathcal{M}}' \models \psi. \tag{4.6}$$

We examine the modified model $\mathcal{N}[\overline{v}' := \mathcal{N}(\overline{v}_0)]$, where $\mathcal{N}$ is defined by

$$\mathcal{N} := \mathcal{M} \cup \overline{\mathcal{M}}'[\overline{v}_0 := (\mathcal{M} \cup \overline{\mathcal{M}}'(\overline{v}'))].$$

The model $\mathcal{N}[\overline{v}' := \mathcal{N}(\overline{v}_0)]$ has the property that it is equal to $\mathcal{M} \cup \overline{\mathcal{M}}'$ except for the variables in $\overline{v}_0$:

$$\begin{aligned}
\mathcal{N}[\overline{v}' := \mathcal{N}(\overline{v}_0)](x) &= \begin{cases} \mathcal{N}(\overline{v}_0) & \text{if } x = \overline{v}' \\ \mathcal{N}(x) & \text{if } x \neq \overline{v}' \end{cases} \\
&= \begin{cases} \mathcal{M} \cup \overline{\mathcal{M}}'(\overline{v}') & \text{if } x = \overline{v}' \\ \mathcal{M} \cup \overline{\mathcal{M}}'(x) & \text{if } x \neq \overline{v}' \wedge x \neq \overline{v}_0 \\ \mathcal{M} \cup \overline{\mathcal{M}}'(\overline{v}') & \text{if } x \neq \overline{v}' \wedge x = \overline{v}_0 \end{cases} \\
&= \begin{cases} \mathcal{M} \cup \overline{\mathcal{M}}'(x) & \text{if } x \neq \overline{v}_0 \\ \mathcal{M} \cup \overline{\mathcal{M}}'(\overline{v}') & \text{if } x = \overline{v}_0. \end{cases}
\end{aligned}$$

Since ψ does not contain the variable $\overline{v}_0$, which has been introduced as fresh variable, we conclude from (4.6)

$$\mathcal{N}[\overline{v}' := \mathcal{N}(\overline{v}_0)] \models \psi.$$

To that we apply the substitution lemma (e.g., [AdBO09]) and get $\mathcal{N} \models \psi^{\overline{v}_0}_{\overline{v}'}$. The model $\mathcal{N}$ is equal to $\mathcal{M}$ except for $\overline{v}_0$, which is not contained in Δ and Γ by

the side-condition of the rule, and for primed symbols, which are not relevant to interpret Δ and Γ. So, from $\mathcal{M} \not\models \Delta \vdash \Gamma$ we infer $\mathcal{N} \not\models \Delta \vdash \Gamma$ and, with premise (4.5) we conclude $\mathcal{N} \models \delta^{\overline{v}_0}_{\overline{v}}$. By applying the substitution lemma again, we infer

$$\mathcal{M} \cup \overline{\mathcal{M}}' \models (\delta^{\overline{v}_0}_{\overline{v}})^{\overline{v}'}_{\overline{v}_0}.$$

Since δ does not contain $\overline{v}_0$, this is equivalent to $\mathcal{M} \cup \overline{\mathcal{M}}' \models \delta^{\overline{v}'}_{\overline{v}}$, in which the formula only contains primed variables, because the side-condition of rule (box step$_\delta$) demands that *all* unprimed system variables in δ are replaced. Thus,

$$\mathcal{M}_{|Const} \cup \overline{\mathcal{M}}' \models \delta^{\overline{v}'}_{\overline{v}},$$

where $\mathcal{M}_{|Const}$ is the model $\mathcal{M}$ reduced to constant symbols. This is by definition of $\overline{\mathcal{M}}'$—defined in Sect. 3.1.1 by $\mathcal{M}'(x') = \mathcal{M}(x)$—equivalent[2] to $\mathcal{M}_{|Const} \cup \overline{\mathcal{M}} \models \delta$. Since $\mathcal{M}$ and $\overline{\mathcal{M}}$ are from the same interpretation $\mathcal{I}$ both models coincide on the constant symbols in *Const* (according to the semantics of processes from Sect. 3.1.1). So, we infer $\overline{\mathcal{M}} \models \delta$, which concludes the proof for the first case of this rule.

In the second case, the argument is dual to the first case. We consider $\mathcal{M}$ with $\mathcal{M} \not\models \Delta \vdash \Gamma$ and $\mathcal{M} \models (\forall \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}} \vdash)$, i.e., $\mathcal{M} \models \neg(\forall \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \Rightarrow \delta^{\overline{v}_0}_{\overline{v}})$, which is equivalent to $\mathcal{M} \models \exists \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \wedge \neg\delta^{\overline{v}_0}_{\overline{v}}$. So, we can assume

$$\forall \mathcal{M} : \mathit{Model} \text{ if } \mathcal{M} \not\models \Delta \vdash \Gamma \text{ then } \mathcal{M} \models \exists \overline{v}_0 \bullet \psi^{\overline{v}_0}_{\overline{v}'} \wedge \neg\delta^{\overline{v}_0}_{\overline{v}} \tag{4.7}$$

and need to prove $\mathcal{M} \models \neg[a \bullet \psi]\delta$, which can be reformulated to $\mathcal{M} \models \langle a \bullet \psi\rangle\neg\delta$. Analogously to the first case, we consider the possible interpretations of $[\![a \bullet \psi]\!]\mathcal{M}$ but we now need to show that there actually is an interpretation with terminating model satisfying δ. This is the case if there is an model $\overline{\mathcal{M}}$ with

$$\mathcal{M} \cup \overline{\mathcal{M}}' \models \psi \text{ and } \overline{\mathcal{M}} \models \neg\delta. \tag{4.8}$$

Without loss of generality, let $\overline{v}_0$ be a fresh symbol with a valuation $\mathcal{M}(\overline{v}_0)$ such that

$$\mathcal{M} \models \psi^{\overline{v}_0}_{\overline{v}'} \wedge \neg\delta^{\overline{v}_0}_{\overline{v}}, \tag{4.9}$$

which is possible because of (4.7). We define $\overline{\mathcal{M}} := \mathcal{M}[\overline{v} := \mathcal{M}(\overline{v}_0)]$ and show that this $\overline{\mathcal{M}}$ satisfies the properties of (4.8):

$$\begin{aligned} (4.9) &\Rightarrow \mathcal{M} \models \psi^{\overline{v}_0}_{\overline{v}'} \\ \{\text{subst. lemma}\} &\Rightarrow \mathcal{M}[\overline{v}' := \mathcal{M}(\overline{v}_0)] \models \psi \\ \{\text{Def. of } \overline{\mathcal{M}}\} &\Rightarrow \mathcal{M} \cup \overline{\mathcal{M}}' \models \psi, \end{aligned}$$

[2] The substitution lemma is implicitly used again for this argument.

where for the last implication we have used that $\mathcal{M}$ does not contain primed symbols and that all primed symbols of ψ are covered by $\overline{v}'$.

$$\begin{aligned} (4.9) &\Rightarrow \mathcal{M} \models \neg\delta^{\overline{v}_0}_{\overline{v}} \\ \{\text{subst. lemma}\} &\Rightarrow \mathcal{M}[\overline{v} := \mathcal{M}(\overline{v}_0)] \models \neg\delta \\ \{\text{Def. of } \overline{\mathcal{M}}\} &\Rightarrow \overline{\mathcal{M}} \models \neg\delta \end{aligned}$$

This proves (4.4) and by this concludes the proof for rule (box step$_\delta$).

Note that this proves the rule to be sound but not locally sound. Recall into memory that local soundness means that every model of the premise is also model of the conclusion. Indeed, the rule is not locally sound, which can be seen by a simple counterexample: consider the dCSP formula $[a \bullet x' = 1](x = 0)$ and a model $\mathcal{M}$ with $\mathcal{M}(v) = 0$. For this model the premise of rule (box step$_\delta$) is valid if we use v as replacement symbol:

$$\begin{aligned} &\mathcal{M} \models (x' = 1)^{v}_{x'} \Rightarrow (x = 0)^{v}_{x} \\ \Rightarrow\ &\mathcal{M} \models (v = 1) \Rightarrow (v = 0) \end{aligned}$$

but $\mathcal{M} \not\models [a \bullet x' = 1](x = 0)$, so the rule is not locally sound.

Rule (box step): Let $\mathcal{M} \models \varphi$ and $\mathcal{M} \models [a \bullet \psi]\varphi$. We need to prove that for every model in the interpretation $\langle \mathcal{M}, a, \overline{\mathcal{M}}, \checkmark, \overline{\mathcal{M}} \rangle \in [\![a \bullet \psi]\!]\mathcal{M}$ the formula φ is valid. For $\mathcal{M}$ this is true due to the first premise. The second premise $\mathcal{M} \models [a \bullet \psi]\varphi$ says that for every terminating model of interpretations in $[\![a \bullet \psi]\!]\mathcal{M}$ the formula φ holds. So, φ also holds for the terminating model $\overline{\mathcal{M}}$ and thus $\mathcal{M} \models [a \bullet \psi]\Box\varphi$.

Rule (parallel$_\Box$): This is a derived rule, because the process $P \parallel_A Q$ is equivalent to a choice of processes. Thus, we can apply rule (process equivalence) to replace the parallel composition by choices, and then apply rule (box choice) to resolve the choices into the conjunctions of the premise of rule (parallel$_\Box$).

Hence, we show that $P \parallel_A Q$ is actually equivalent to a choice of processes in the desired manner. Let P and Q be processes as defined in the side-condition of rule (parallel$_\Box$). We show that $P \parallel_A Q$ is equivalent to

$$\begin{aligned} R := {} & a_1 \bullet (\varphi_1 \wedge \psi_1) \rightarrow (P_1 \parallel_A Q_1) \\ & \Box\ \ldots \\ & \Box\ a_{n_0} \bullet (\varphi_{n_0} \wedge \psi_{n_0}) \rightarrow (P_{n_0} \parallel_A Q_{n_0}) \\ & \Box\ b_1 \bullet \varphi_{n+1} \rightarrow (P_{n+1} \parallel_A Q) \\ & \Box\ \ldots \\ & \Box\ c_{\overline{m}} \bullet \psi_{m+\overline{m}} \rightarrow (P \parallel_A Q_{m+\overline{m}}). \end{aligned}$$

The argument is by case distinction over the possible transitions of $P \parallel_A Q$:

- a_i for $i \in 1..n_0$: All these a_i are in A, and P as well as Q can engage in the event a_i. Thus, a synchronised transition

  $$a_i \bullet (\varphi_i \wedge \psi_i) \to (P_i \parallel_A Q_i)$$

  is possible. According to the semantics of synchronisation in CSP with data, the constraints of both events are conjugated.
- a_i and $\overline{a}_i$ for $i > n_0$: Because of $a_i \in A$, the parallel composition over alphabet A demands synchronisation on these events, but each event is only possible in one of the components. So, no transition is possible for these events.
- b_i for $i \in 1..\overline{n}$: As b_i is not in A, a b_i transition of process P is possible, whereas Q does nothing:

  $$b_i \bullet (\varphi_{n+i}) \to (P_{n+i} \parallel_A Q).$$

- c_i for $i \in 1..\overline{m}$: Symmetrically to the previous case, a c_i transition is possible for Q:

  $$c_i \bullet (\psi_{m+i}) \to (P \parallel_A Q_{m+i}).$$

These are exactly all possible transitions for $P \parallel_A Q$ and of the choice process R. Thus, the processes are equivalent in trace and interpretation semantics.

Rule (assumption axiom): It is to be shown that each terminating model of interpretations in $[\![\mathtt{Proc}^{\infty}_{\backslash A,V}]\!]\mathcal{M}$ satisfies formula δ. This is trivially true, because the process $\mathtt{Proc}^{\infty}_{\backslash A,V}$ does not contain the process `Skip` as sub-process and, thus, there is no terminating model.

Rule (box assumption$_\delta$): We assume that

$$\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \varphi \vdash \delta \tag{4.10}$$

and need to show that for all sets of formulae Δ and Γ also

$$\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \Delta, \psi \vdash [\mathtt{Proc}_{\backslash A,V} \bullet F]\delta, \Gamma$$

holds. To this end, we must prove that for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ and all interpretations $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ with terminating model $\overline{\mathcal{M}}$ the formula δ is valid, i.e., $\overline{\mathcal{M}} \models \delta$. Due to the side-condition of rule (box assumption$_\delta$), we can infer that $\overline{\mathcal{M}} \models \varphi$ and with the precondition (4.10) we conclude $\overline{\mathcal{M}} \models \delta$.

Note that we actually need a proposition for all models $\mathcal{M}$ in the premise (4.10). This is the reason for the special shape of rule (box assumption$_\delta$), where in the conclusion the context formulae Δ and Γ are considered but not in the premise. With a premise with context

$$\Delta, \varphi \vdash \delta, \Gamma$$

the rule would become inaccurate, because this premise implies

$$\forall \mathcal{M} \in \mathit{Model} : \Delta, \varphi \vdash \delta, \Gamma$$

instead of (4.10). Hence, it is possible that Γ holds for $\overline{\mathcal{M}}$ but not δ—which is not the case with the premise in (4.10).

Rule (assumption$_\Box$): We need to show that for all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ and interpretations $\mathcal{I} \in [\![\mathtt{Proc}^{(\infty)}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ the property $\mathcal{I} \models \Box\varphi$ is valid, which is exactly the statement of the side-condition of the rule.

Rule (box loop gen): The process P is recursively defined by $P \overset{c}{=} F(P)$, i.e., it is given by the fixed point of F. We first define the set of interpretations

$$L := \left\{ [\![Q]\!] \,|\, \forall \mathcal{N} \in \mathit{Model} : \left((\mathcal{N} \models \varphi_{in}(\overline{y})) \Rightarrow [\![Q]\!]\mathcal{N} \subseteq [\![\gamma(\overline{y})]\!]\right) \right\},$$

where we abbreviate $[\![Q]\!] := \bigcup_{\mathcal{M}} [\![Q]\!]\mathcal{M}$. Since γ is of the shape $\Box\delta$ or δ, as stated in the side-condition of rule (box loop gen), the constraint $[\![Q]\!]\mathcal{N} \subseteq [\![\gamma(\overline{y})]\!])$ is equivalent to $\mathcal{N} \models [Q]\gamma(\overline{y})$. The idea of this set L is that it contains all traces of "good" processes P for that the desired property $\mathcal{N} \models [P]\gamma(\overline{y})$ holds in the case that $\mathcal{N}$ is a model satisfying $\varphi_{in}(\overline{y})$.

We prove the soundness of the rule in three steps:

1. We prove that the set L is a complete lattice with respect to the $\subseteq$-order.
2. On this set, we define a function f that is the semantical counterpart to the process function $F(P)$.
3. Then, we show that f is monotone (and continuous) and, thus, it has a fixed point in L, which corresponds to the fixed point of $F(P)$.

With this last argument, the proof is completed, because $F(P)$ then has the desired property by construction of L. We now go through the three proof steps:

1. The set of non-empty, prefix closed set of traces of a CSP process is a complete lattice wrt. the $\subseteq$-order [Ros98]. We need to show that, likewise, the set L is a complete lattice for the $\subseteq$-order. Let X be a non-empty subset of L, i.e., for each $[\![Q_i]\!] \in X$ it holds $\forall \mathcal{N} : (\mathcal{N} \models \varphi_{in}(\overline{y}) \Rightarrow [\![Q_i]\!]\mathcal{N} \subseteq [\![\gamma(\overline{y})]\!])$. The least upper bound of X, written $\bigsqcup X$, is given by $\bigcup X$, that is non-empty again. $\bigcup X$ is actually in L, because it is in the semantics of the choice process $\Box_i\, Q_i$, which is given by

$$[\![\Box_i\, Q_i]\!]\mathcal{M} = \bigcup_i [\![Q_i]\!]\mathcal{M}.$$

Thus, $\bigsqcup X = \bigcup X = [\![\Box_i\, Q_i]\!]$. For every model $\mathcal{M}_0$ with $\mathcal{M}_0 \models \varphi_{in}(\overline{y})$ it holds $[\![Q_i]\!]\mathcal{M}_0 \subseteq [\![\gamma(\overline{y})]\!]$. Hence,

$$\bigcup_i [\![Q_i]\!]\mathcal{M} \subseteq [\![\gamma(\overline{y})]\!] \text{ and by this } [\![\Box_i\, Q_i]\!]\mathcal{M}_0 \subseteq [\![\gamma(\overline{y})]\!]$$

and we conclude $\bigsqcup X = [\![\Box_i\, Q_i]\!] \in L$—the least upper bound exists and is in L.

The greatest lower bound of X, written $\bigsqcap X$, is given by $\bigcap X$. This is non-empty again because every $[\![Q_i]\!] \in X$ contains the subset $\{\langle \mathcal{M} \rangle \mid \mathcal{M} \in \mathit{Model}\}$, i.e., an empty interpretation, corresponding to an empty run of the LTS of Q_i, for every initial model. For this reason, this subset is the $\bot$-element of L. $\bigcap X$ is actually in L because for every $\mathcal{M}_0 \models \varphi_{\mathit{in}}(\overline{y})$

$$\bigcap X_{|\mathcal{M}_0} \subseteq [\![Q_i]\!]\mathcal{M}_0 \subseteq [\![\gamma(\overline{y})]\!],$$

in which $\bigcap X_{|\mathcal{M}_0}$ is the restriction to interpretations starting with model $\mathcal{M}_0$. Moreover, $\bigcap X$ is equal to the semantics of the process that synchronises on all events and their constraints, denoted by $\|_i\ Q_i$. By this, $[\![\|_i\ Q_i]\!]$ contains exactly the interpretations that are allowed by all processes Q_i. So,

$$\bigsqcap X = \bigcap_i [\![Q_i]\!] = [\![\|_i\ Q_i]\!] \in L.$$

By this means, L is a complete lattice with the identified $\bot$ and the $\top$-element L.

2. We now define a function f on L, which is the semantical counterpart of the process function $F(P)$, with the basic idea that the semantics of the fixed point of $F(P)$ is a fixed point of f. Let f be a function $f : L \to L$ with

$$f([\![Q]\!]) := [\![F(Q)]\!].$$

That is, f maps the set of interpretations for Q to the semantics of $F(Q)$. This function is well-defined if we can show that $[\![F(Q)]\!]$ is actually an element of L. We prove this by taking the premises of rule (box loop gen) into account:

We consider a process Q with $[\![Q]\!] \in L$. Therefore,

$$\begin{aligned} &\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \varphi_{\mathit{in}}(\overline{y}) \Rightarrow [\![Q]\!]\mathcal{M} \subseteq [\![\gamma(\overline{y})]\!] \\ \Leftrightarrow &\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \varphi_{\mathit{in}}(\overline{y}) \Rightarrow [Q]\gamma(\overline{y}) \\ \Leftrightarrow &\forall\, \mathcal{M} \in \mathit{Model} : \mathcal{M} \models \forall\, \overline{x} : \varphi_{\mathit{in}}(\overline{x}) \Rightarrow [Q\tfrac{\overline{x}}{\overline{y}}]\gamma(\overline{x}) \end{aligned} \tag{4.11}$$

The last equivalence is correct, because $\varphi_{\mathit{in}}(\overline{x})$ and $\gamma(\overline{x})$ contain by assumption only system variables $\overline{x}$. To show that $[\![F(Q)]\!]$ is in L, let $\mathcal{M}_0$ be a model with $\mathcal{M}_0 \models \varphi_{\mathit{in}}(\overline{y})$. To this and to (4.11), we apply the right premise of the rule,

$$\varphi_{\mathit{in}}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{\mathit{in}}(\overline{x}) \Rightarrow [Q\tfrac{\overline{x}}{\overline{y}}]\gamma(\overline{x})) \vdash [F(Q)]\gamma(\overline{y}),$$

and get $\mathcal{M}_0 \models [F(Q)]\gamma(\overline{y})$, which is equivalent to $[\![F(Q)]\!]\mathcal{M}_0 \subseteq [\![\gamma(\overline{y})]\!]$. That is, $f([\![Q]\!]) = [\![F(Q)]\!] \in L$.

3. According to Thm. 3.2.1, all considered CSP operators are continuous for the $\subseteq$-order on the interpretation semantics. Parallel compositions are replaced by equivalent choices over processes (Cor. 3.3.3). Thus, we infer that $f(\cdot)$ is also continuous and monotonic. With the Knaster-Tarski theorem [Kna28, Tar55], we conclude that f has a least fixed point in L, i.e.,

$$\mu\, X.f(X) \in L. \tag{4.12}$$

The semantics of the process P, declared by $P \stackrel{c}{=} F(P)$, is identified with the least fixed point of the interpretations of $F(P)$, i.e.,

$$[\![P]\!] = [\![\mu\, X.F(X)]\!] := \mu[\![X]\!].[\![F(X)]\!] = \mu\, X.f(X).$$

Thus, with (4.12) $[\![P]\!] \in L$.

With this observation we can finally conclude the soundness of the rule: Consider a model $\mathcal{M}$ fulfilling the left premise, $\mathcal{M} \models \Delta \vdash \varphi_{in}(\overline{y}), \Gamma$. If $\mathcal{M} \models \Delta \vdash \Gamma$, the conclusion of the rule follows directly. So we assume $\mathcal{M} \models \varphi_{in}(\overline{y})$. Due to $[\![P]\!] \in L$ we infer from the definition of L that if $\mathcal{M} \models \varphi_{in}(\overline{y})$, then $[\![P]\!]\mathcal{M} \subseteq [\![\gamma(\overline{y})]\!]$, which is equivalent to the desired $\mathcal{M} \models [P]\gamma(\overline{y})$.

Rule (box loop): This rule is a consequence of (box loop gen), that can be derived within the sequent calculus. We give the proof tree here. Let $P \stackrel{c}{=} Q \mathbin{⨾} P$. We set $F(P) = Q \mathbin{⨾} P$, $\varphi_{in}(\overline{y}) = \varphi_{in}$, and $\gamma(\overline{y}) = \gamma$ to be able to apply rule (box loop gen). The proof tree demonstrates the application of the induction rule (box loop gen), as well as the application of rule (process step).

In the first part of the proof tree, the rule (box loop gen) is applied. The open goal $\Delta \vdash \varphi_{in}(\overline{y}), \Gamma$ is the first premise of the rule (box loop). In the next step, rule (sequence$_\Box$) is applied, assuming that γ begins with the $\Box$-operator. If this is not the case, i.e., in the δ-case, then rule (sequence$_\delta$) is to be applied, the next intermediate step (and right) and sub-goal (G1) are omitted, and the proof continues in exactly the same way with goal (G2).

$$\dfrac{\Delta \vdash \varphi_{in}(\overline{y}), \Gamma \qquad \dfrac{\dfrac{(G1) \qquad (G2)}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P\tfrac{\overline{x}}{\overline{y}}]\gamma(\overline{x})) \vdash [Q]\gamma(\overline{y}) \wedge [Q][P]\gamma(\overline{y})}\text{(and right)}}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P\tfrac{\overline{x}}{\overline{y}}]\gamma(\overline{x})) \vdash [Q \mathbin{⨾} P]\gamma(\overline{y})}\text{(sequence}_\Box\text{)}}{\Delta \vdash [P]\gamma(\overline{y}), \Gamma}\text{(box loop gen)}$$

Sub-goal (G1) can be closed with $\varphi_{in}(\overline{y}) \vdash [Q]\gamma(\overline{y})$, which is a further premise of rule (box loop):

$$\dfrac{\varphi_{in}(\overline{y}) \vdash [Q]\gamma(\overline{y})}{\underset{(G1)}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P\tfrac{\overline{x}}{\overline{y}}]\gamma(\overline{x})) \vdash [Q]\gamma(\overline{y})}}\text{(weakening left)}$$

We apply the cut rule to goal (G2) and make a case distinction over the last premise of rule (box loop), $\varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y})$.

$$\dfrac{(\text{C1}) \qquad\qquad (\text{C2})}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [Q][P]\gamma(\overline{y})}\;(\text{cut})$$
$$(\text{G2})$$

For the first case, we remove unneeded formulae and resolve the implication to end up with the right premise of rule (box loop).

$$\dfrac{\dfrac{\dfrac{\dfrac{\varphi_{in}(\overline{y}) \vdash [Q]\varphi_{in}(\overline{y})}{\vdash \varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y})}\;(\text{implication right})}{\vdash [Q][P]\gamma(\overline{y}), \varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y})}\;(\text{weakening right})}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [Q][P]\gamma(\overline{y}), \varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y})}\;(\text{weakening left})}{}$$
$$(\text{C1})$$

To close the proof, we finally apply the induction hypothesis (by instantiating the $\forall$-constraint). In addition, we make use of the premise $\varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y})$ by applying the rule (process step) to consume the Q process.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{(\text{axiom})}{\varphi_{in}(\overline{v}_0), \vdash [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0), \varphi_{in}(\overline{v}_0)} \qquad \overset{(\text{axiom})}{\varphi_{in}(\overline{v}_0), [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0) \vdash [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0)}}{\varphi_{in}(\overline{v}_0), \varphi_{in}(\overline{v}_0) \Rightarrow [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0) \vdash [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0)}\;(\text{implication left})}{\varphi_{in}(\overline{v}_0), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0)}\;(\text{all left})}{\varphi_{in}(\overline{v}_0), \varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [P^{\overline{v}_0}_{\overline{y}}]\gamma(\overline{v}_0)}\;(\text{weakening left})}{[Q]\varphi_{in}(\overline{y}), \varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [Q][P]\gamma(\overline{y})}\;(\text{process step})$$

$$\dfrac{\overset{(\text{axiom})}{\varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [Q][P]\gamma(\overline{y}), \varphi_{in}(\overline{y})} \qquad \diagdown}{\varphi_{in}(\overline{y}) \Rightarrow [Q]\varphi_{in}(\overline{y}), \varphi_{in}(\overline{y}), \forall\, \overline{x} \bullet (\varphi_{in}(\overline{x}) \Rightarrow [P^{\overline{x}}_{\overline{y}}]\gamma(\overline{x})) \vdash [Q][P]\gamma(\overline{y})}\;(\text{implication left})$$
$$(\text{C2})$$

This finishes the proof of rule (box loop).

As mentioned above, if γ is a δ-formula, the proof tree remains exactly the same except for the omission of one application of (and right) and the omission of sub-goal (G1). Interestingly, without sub-goal (G1) the proof does not depend on the premise $\varphi_{in}(\overline{y}) \vdash [Q]\gamma(\overline{y})$ anymore. Thus, it can be omitted in the δ-case. By this, no premise of the rule contains γ, which seems weird at first sight. But, P is a non-terminating process and, thus, every formula is valid after every terminating run of P. This is correctly revealed by this proof tree. The consequence is that rule (box loop) is indeed valid for the δ-case but does not make much sense for δ-expressions.

Rule ($\Box$-introduction): Let $\mathcal{M} \models [P]\Box\varphi$, i.e., for all $\mathcal{I} \in [\![P]\!]\mathcal{M}$ hold $\mathcal{I} \models \Box\varphi$, thus, $\forall\, t : \mathcal{I}(t) \models \varphi$. If $\mathcal{I}$ has a terminating model, i.e., $\mathcal{I}(t_0) = \overline{\mathcal{M}}$ for a point in time t_0, then also $\overline{\mathcal{M}} \models \varphi$.

Rule (process step): For all models $\mathcal{M}$ with $\mathcal{M} \not\models \Delta \vdash \Gamma$ the premise

$$\Delta, \delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0}, \Gamma$$

allows us to deduce $\mathcal{M} \models \delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0}$. If $\mathcal{M} \models \Delta \vdash \Gamma$, then the conclusion directly holds for this $\mathcal{M}$.

Let $\mathcal{M}$ now be such a model with $\mathcal{M} \not\models \Delta \vdash \Gamma$ and $\mathcal{M} \models \delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0}$. In a first step, we show that for all models $\mathcal{N}$ coinciding with $\mathcal{M}$ on all constant symbols, $\mathcal{N}_{|Const} = \mathcal{M}_{|Const}$, the property $\mathcal{N} \models \delta \vdash \epsilon$ holds. This is valid, because $\overline{v}_0$ does by assumption not occur in Δ or Γ, and $\delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0}$ holds for arbitrary valuations of $\overline{v}_0$:

$$\begin{aligned} & \forall \mathcal{N} : \mathit{Model} \mid \mathcal{N}_{|Const} = \mathcal{M}_{|Const}, \mathcal{N}(\overline{y}) = \mathcal{M}(\overline{y}) \bullet \mathcal{N} \not\models \Delta \vdash \Gamma && (4.13)\\ \Rightarrow\ & \forall \mathcal{N} : \mathit{Model} \mid \mathcal{N}_{|Const} = \mathcal{M}_{|Const}, \mathcal{N}(\overline{y}) = \mathcal{M}(\overline{y}) \bullet \mathcal{N} \models \delta_{\overline{y}}^{\overline{v}_0} \vdash \epsilon_{\overline{y}}^{\overline{v}_0} \\ \Rightarrow\ & \forall \mathcal{N} : \mathit{Model} \mid \mathcal{N}_{|Const} = \mathcal{M}_{|Const}, \mathcal{N}(\overline{y}) = \mathcal{M}(\overline{y}) \bullet \mathcal{N}[\overline{y} = \mathcal{N}(\overline{v}_0)] \models \delta \vdash \epsilon. && (4.14) \end{aligned}$$

Formula (4.13) holds because $\mathcal{M} \not\models \Delta \vdash \Gamma$, and $\mathcal{M}$ and $\mathcal{N}$ agree on all constant symbols and system variables $\overline{y}$. Formula (4.14) follows with the substitution lemma. In particular, (4.14) holds for every possible valuation $\mathcal{N}(\overline{v}_0)$ (since $\overline{v}_0$ has been introduced as fresh symbol and is not further constrained in Δ or Γ) and is independent of $\mathcal{N}$'s valuation for $\overline{y}$. Thus, we can further infer

$$\begin{aligned} & (4.14) \\ \Leftrightarrow\ & \forall \mathcal{N} \mid \mathcal{N}_{|Const} = \mathcal{M}_{|Const} \bullet \mathcal{N}[\overline{y} = \mathcal{N}(\overline{v}_0)] \models \delta \vdash \epsilon && (4.15)\\ \Leftrightarrow\ & \forall \mathcal{N} \mid \mathcal{N}_{|Const} = \mathcal{M}_{|Const} \bullet \mathcal{N} \models \delta \vdash \epsilon && (4.16) \end{aligned}$$

With this we show $\mathcal{M} \models [Q]\delta \vdash [Q]\epsilon$. So, assuming $\mathcal{M} \models [Q]\delta$, we get from the semantics of $[Q]\delta$ that for all interpretations

$$\langle \mathcal{M}, a, \mathcal{M}_1, \ldots, \mathcal{M}_n, \checkmark \rangle \in [\![Q]\!]\mathcal{M}$$

the constraint $\mathcal{M}_n \models \delta$ holds. The semantics (cf. Sect. 3.1.1) demands that no symbols from *Const* change within an interpretation. Therefore, $\mathcal{M}_{n|Const} = \mathcal{M}_{|Const}$. With (4.16) we conclude $\mathcal{M}_n \models \epsilon$ and, thus, the desired $\mathcal{M} \models [Q]\epsilon$.

Diamond cases: The proofs for the diamond cases of the rules are predominantly completely dual to the box cases, because the rule schemata are symmetric, i.e.,

they can be applied to both sides of the sequent symbol (negated and non-negated formulae), and because of the equality

$$[P]\gamma = \neg\langle P\rangle\neg\gamma. \tag{4.17}$$

The diamond cases of the rules (prefix$_\delta$), (prefix$_\diamond$), (diamond choice), (sequence$_\delta$), (sequence$_\diamond$), (diamond step$_\delta$), (diamond step) can be directly proven by this duality. For example, the soundness of (sequence$_\delta$) can be shown by the transformation

$$\begin{aligned}
&\mathcal{M} \models \langle P_1\rangle\langle P_2\rangle\delta \\
\{(4.17)\} \Leftrightarrow\ &\mathcal{M} \models \neg[P_1]\neg\langle P_2\rangle\delta \\
\{(4.17)\} \Leftrightarrow\ &\mathcal{M} \models \neg[P_1][P_2]\neg\delta \\
\{(*)\} \Leftrightarrow\ &\mathcal{M} \models \neg[P_1 \fatsemi P_2]\neg\delta \\
\{(4.17)\} \Leftrightarrow\ &\mathcal{M} \models \langle P_1 \fatsemi P_2\rangle\delta,
\end{aligned}$$

in which equivalence $(*)$ follows from the box case of rule (sequence$_\delta$)[3].

The rules (interleaving$_\diamond$) and (sync$_\diamond$) can be combined to a rule, which is dual to (parallel$_\Box$). For this reason, (interleaving$_\diamond$) and (sync$_\diamond$) are in this split form not locally sound. Rule (skip$_\diamond$) is sound, because nothing prevents `Skip` from terminating and `Skip` does not change the state space. And finally, the side-conditions of the rules (diamond assumption$_\delta$) and (assumption$_\diamond$) directly imply the soundness of the rules. □

Theorem 4.4.2 (Incompleteness)
The calculus as presented in Sect. 4.3 is not complete, i.e., we cannot derive every valid formula in the calculus.

Proof. This is a direct consequence of the integration of an arbitrary temporal logic to constrain unknown processes. By this, we can choose an undecidable logic like the full DC [ZH04] and, thus, cannot resolve the constraints of those unknown processes in every case.

With the restriction to decidable logics for unknown parts, our calculus is still incomplete, because dCSP includes first-order arithmetic. By this, it is a consequence of Gödel's *Incompleteness Theorem* [Göd31] that our calculus can also not be complete. However, for similar incomplete calculi one can often state *relative completeness* results, showing that the non-derivable first-order formulae are the only cause that inhibits completeness of the calculus. That is, one has to prove that for any valid sequent $\Delta \vdash \Gamma$, there is a set Δ_{FO} of valid first-order formulae such that $\Delta_{FO}, \Delta \vdash \Gamma$ is derivable in the calculus (cf. [Har79, BHS07]). Even though it is likely that a similar result can be established for our calculus, completeness of the calculus is so far an open issue. □

[3]The local soundness of the rule for the box case actually implies that conclusion and premise are equivalent.

4.5. Embedding of a Real-Time Logic

The calculus presented in the previous section is generic in terms of the underlying real-time logic to specify assumptions on unknown parts. For this reason, the side-conditions of the rules (box assumption$_\delta$) up to (box assumption$_\delta$) need to be verified in the applied real-time logic. Thus, we now give an example embedding of the real-time logic Duration Calculus (DC) in order to verify the side-conditions of the rules over unknown parts. We choose the DC because it is used in our case studies as language for assumptions over unknowns.

We show how to use the model checking approach from [MFHR08] to reduce formulae with unknown processes and DC formulae by applying the rules (box assumption$_\delta$) – (assumption$_\diamond$) automatically.

4.5.1. Checking the Side-Conditions for the Box Operator

We recall that if we want to reduce a process using rule (box assumption$_\delta$), we need to check the side-condition:

> For all models $\mathcal{M}$ with $\mathcal{M} \models \psi$ and and all interpretations
> $\mathcal{I} \in [\![\mathtt{Proc}_{\setminus A,V} \bullet F]\!]\mathcal{M}$ the formula φ holds: $\mathcal{I} \models \varphi$.

Since DC formulae are defined in the same semantical domain, i.e., the semantics of DC formulae (cf. Sect. 2.1.3) is given by mappings from a time domain into the set of all models $\mathcal{I} : \mathit{Time} \to \mathit{Model}$, we can express the negation of this property with the DC formula

$$(\lceil\psi\rceil \frown \text{true}) \wedge F \wedge \bigwedge_{ev \in A} \boxminus\, ev \wedge (\text{true} \frown \updownarrow\checkmark \frown \lceil\neg\varphi\rceil), \tag{4.18}$$

where ψ and φ are time-dependent state expressions and the variables in V are global constants. Runs that are terminated must explicitly perform a $\checkmark$-event such that the terminating model can be checked with DC. This causes no problems when checking the DC formulae against PEA, where such a $\checkmark$ can be added to every transition reaching a terminating location (e.g., a location representing the Ω process when the PEA is generated from a CSP process)[4]. The unsatisfiability of formula (4.18) can be verified automatically using the PEA construction for DC formulae from [Hoe06, MFHR08] (cf. Sect. 2.3.2).

Remark 4.5.1. Actually, this verification approach [Hoe06, MFHR08], which uses a power set construction to obtain a PEA that equivalently represents a DC formula, does not consider DC formulae with global variables, i.e., variables that are *not* time-dependent. Anyway, it is easy to extend the power set construction for DC with

[4] Another possibility without an explicit $\checkmark$, which often can be applied, is to check an invariant condition $\mathcal{I} \models \Box\varphi_{in}$ instead, that implies φ.

constant variables V by adding the constraint

$$\bigwedge_{v \in V} v' = v$$

to every transition of the resulting PEA. This way, the variables in V can initially be set to any value (if not further restricted), but may never be changed during the run of the resulting automaton.

Analogously, the side-condition of rule (assumption$_\square$) is equivalent to the unsatisfiability of

$$(\lceil \psi \rceil ^\frown \text{true}) \wedge F \wedge \bigwedge_{ev \in A} \boxminus ev \wedge (\Diamond \lceil \neg \varphi \rceil), \tag{4.19}$$

Remark 4.5.2. Property (4.19) implies property (4.18), so often it suffices to check only (4.19) to verify the side-conditions of both (assumption$_\square$) and (box assumption$_\delta$).

We prove the correctness of the stated equivalences:

Theorem 4.5.3
The DC formula (4.18) *is unsatisfiable if and only if for all models $\mathcal{M} \models \psi$ and interpretations $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$ the terminating model $\overline{\mathcal{M}}$ of $\mathcal{I}$ is model of φ: $\overline{\mathcal{M}} \models \varphi$.*

Proof. We show both directions:

"⇒" We consider an arbitrary model $\mathcal{M}$ with $\mathcal{M} \models \psi$ and an interpretation $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$. We need to show for the terminating model $\overline{\mathcal{M}} \models \varphi$. Due to the semantics of $[\![\mathtt{Proc}_{\backslash A,V} \bullet F]\!]\mathcal{M}$, we can conclude $\mathcal{I}(0) \models \psi$ and $\mathcal{I} \models F$. From the former, we also get $\mathcal{I} \models \lceil \psi \rceil ^\frown \text{true}$. In the transition rule for $\mathtt{Proc}_{\backslash A,V}$ there are no transitions that changes variables in V. So, $\forall\, t_1, t_2 : \mathit{Time}, v : V \bullet \mathcal{I}(t_1)(v) = \mathcal{I}(t_2)(v)$, which means that all variables in V are global constants in the sense of DC. In addition, the transition rule in 3.1.2 contains no transition for events from A, so we also get $\mathcal{I} \models \bigwedge_{e \in A} \boxminus e$. We summarise our knowledge about $\mathcal{I}$:

$$\mathcal{I} \models (\lceil \psi \rceil ^\frown \text{true}) \wedge F \wedge (\bigwedge_{e \in A} \boxminus e)$$

for global constants V. Due to our assumption that (4.18) is unsatisfiable we can conclude

$$\mathcal{I} \not\models (\text{true} ^\frown \updownarrow\checkmark ^\frown \lceil \neg \varphi \rceil).$$

That is, if $\mathcal{I}$ has a terminating model $\overline{\mathcal{M}}$ then $\overline{\mathcal{M}} \models \varphi$ must be true.

"⇐" We prove the second case by contradiction and assume that (4.18) is satisfiable. That is, there is an interpretation $\mathcal{I}$ with terminating model such that

$$\mathcal{I} \models \lceil \psi \rceil ^\frown \text{true} \tag{4.20}$$

$$\mathcal{I} \models F \tag{4.21}$$

$$\mathcal{I} \models \bigwedge_{e \in A} \boxminus\, e \tag{4.22}$$

$$\mathcal{I} \models \text{true} ^\frown \updownarrow\checkmark ^\frown \lceil \neg \varphi \rceil . \tag{4.23}$$

From (4.23), we conclude for the terminating model $\overline{\mathcal{M}}$ that

$$\overline{\mathcal{M}} \models \neg \varphi \tag{4.24}$$

holds. With (4.20), $\mathcal{I}(0) \models \psi$ is true. Since all variables from V are DC constants in formula (4.18) and due to (4.21) and (4.22), we get

$$\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A, V} \bullet F]\!]\mathcal{I}(0).$$

We can now apply the precondition by which for such interpretations with $\mathcal{I}(0) \models \psi$ the terminating model $\overline{\mathcal{M}}$ is also model of φ. Thus, $\overline{\mathcal{M}} \models \varphi$, which is a contradiction to (4.24). □

The proof that formula (4.19) correctly reflects the side-condition of (assumption$_\Box$) is very similar: we do not need to take care of the terminating model, but examine the desired property φ for every model at arbitrary points in time of the interpretation.

4.5.2. Checking the Side-Conditions for the Diamond Operator

Checking the side-conditions of the diamond operator rules is not as straightforward as for the box operator case. The reason is that the Duration Calculus is a linear-time logic allowing us to verify properties over all initial models and all interpretations of formulae (or its negation, that there is one initial model for that the desired interpretation can be found). But here we need to verify side-conditions of the shape that *for all models* $\mathcal{M}$ with $\mathcal{M} \models \psi$ *there is* an interpretation $\mathcal{I} \in [\![\mathtt{Proc}_{\backslash A, V} \bullet F]\!]\mathcal{M}$ for that the formula $\Diamond\varphi$ holds, $\mathcal{I} \models \Diamond\varphi$. Thus, we need to figure out the existence of a particular run *for every possible initial model* $\mathcal{M}$ that makes the desired property valid. We are basically much more interested in the box case, because our aim is usually to verify safety of real-time systems and not liveness properties (for which the DC may not be the right choice). However, with some effort (and for a restricted set of DC formulae) it is possible to also verify the diamond cases using DC formulae. For the sake of completeness, we sketch the idea of how to verify the side-conditions of the diamond rules here.

As already mentioned, the problem with the diamond case is that we need to find a valid trace or interpretation for every possible initial model $\mathcal{M}$, i.e., models with

$\mathcal{M} \models \psi$. Hence, the straightforward solution to solve the problem by checking the satisfiability of the DC formula

$$(\lceil\psi\rceil^\frown \text{true}) \wedge F \wedge \bigwedge_{ev \in A} \boxminus\, ev \wedge (\mathit{true}^\frown \updownarrow\checkmark^\frown \lceil\varphi\rceil) \tag{4.25}$$

fails, because it only yields that there is at least *one* such interpretation without taking the initial model $\mathcal{M}$ into account. A simple counterexample is (with $\psi \equiv$ true, $F = (\lceil x < 10 \Rightarrow y > 0\rceil)$, and $\varphi \equiv y = 0$)

$$(\lceil x < 10 \Rightarrow y > 0\rceil) \wedge (\mathit{true}^\frown \updownarrow\checkmark^\frown \lceil y = 0\rceil), \tag{4.26}$$

which is satisfiable for $x \geq 10$, but the rule (diamond assumption$_\delta$) requires that there is an interpretation for all initial models $\mathcal{M}$, so even for $\mathcal{M} \models x < 10$, for which (4.26) has no interpretation. Thus, formulae of the shape of (4.25) cannot be used to verify the side-condition of the rule.

The idea to solve this issue is to partition all possible initial models according to the value of ψ and the constraints of F such that all relevant values of these constraints are covered. Then, we verify the validity of the desired side-condition by checking the satisfiability of (4.25) for all possible configurations.

To this end, we restrict ourselves to DC trace formulae F that can be represented as disjunctive normal form of the shape $F \equiv F_1 \vee \ldots \vee F_n$ and for that all possible initial values can be described by constraints ψ_i for $i \in 1..n$ with the property

$$\mathcal{M} \models \psi_i \quad \text{if} \quad \mathcal{M} \in \{\mathcal{I} : [\![F_i]\!] \bullet \mathcal{I}(0)\}$$
$$\psi \Leftrightarrow \psi_1 \vee \ldots \vee \psi_n.$$

These configurations ψ_i must uniquely cover all possible valuations of system variables and constants, which is directly possible for finite data domains and sufficient determined initial constraints ψ, i.e., if ψ describes a finite number of valuations for all relevant symbols (for instance, $\psi \equiv x = 15 \vee x = 3$ if x is the only variable in the considered trace).

For infinite data domains, one has to find a way to cover all possible initial models in an adequate way. The correctness condition is that all interpretations of F_i starting with a model of ψ_i can be slightly modified by exchanging the first model of the interpretation by any other model satisfying ψ_i. This condition is formally defined by

$$\begin{aligned}
&\exists\, \mathcal{M} : \mathit{Model}, \mathcal{I} : [\![F_i \wedge (\mathit{true}^\frown \updownarrow\checkmark^\frown \lceil\varphi\rceil)]\!] \bullet \mathcal{M} \models \psi_i \wedge \mathcal{I}(0) = \mathcal{M} \\
&\qquad\qquad\qquad\qquad \Rightarrow \\
&\quad \forall\, \mathcal{N} : \mathit{Model} \mid \mathcal{N} \models \psi_i \bullet (\exists\, \mathcal{I} \in [\![F_i \wedge (\mathit{true}^\frown \updownarrow\checkmark^\frown \lceil\varphi\rceil)]\!], t_0 \in \mathit{Time} \\
&\qquad \bullet\, \mathcal{I}'(t) = \mathcal{N} \wedge \mathcal{I}(t) = \mathcal{M} \text{ for } t < t_0 \text{ and } \mathcal{I}'(t) = \mathcal{I}(t) \text{ for } t > t_0),
\end{aligned} \tag{4.27}$$

which is satisfied for all configurations ψ_i describing a single valuation for every variable in the DC formula as in the example above. In the case that the constraints

in the DC formulae are arithmetical constraints over real-valued system variables one could use constraints restricting the range of the variables. For instance, the formula in (4.26) could be checked using initial configurations $x \geq 10$ and $y > 0$, by which the latter reveals (4.26) to be unsatisfiable for $y > 0$. Due to our focus on safety properties, we do not go into detail and do not examine here how to find such configurations in general.

Given suitable configurations according to condition (4.27), it can be shown for all $i \in 1..n$

$$(\lceil \psi \wedge \psi_i \rceil ^\frown \text{true}) \wedge F_i \wedge \bigwedge_{ev \in A} \boxminus ev \wedge (true ^\frown \updownarrow \checkmark ^\frown \lceil \varphi \rceil) \text{ is satisfiable} \tag{4.28}$$

iff the side-condition of (diamond assumption$_\delta$) is valid. Variables $v \in V$ in the side-condition are assumed to be global DC variables.

Remark 4.5.4. Without loss of generality, we assume that none of the ψ_i is covered by another one, i.e., there is no index i with $\psi_i \Rightarrow (\bigvee_{j \in \{1..n\} \setminus \{i\}} \psi_j)$. In this case, we could ignore the index i and check (4.28) for all indices but i. In the special case that $\psi_i = \psi_j$ for $i \neq j$ we only need to check the satisfiability of the DC formulae in (4.28) for one of the indices.

The following theorem shows that (4.28) can indeed be used for the verification of the side-condition.

Theorem 4.5.5
The DC formulae in (4.28) *are satisfiable if and only if for all models $\mathcal{M} \models \psi$ there is an interpretation $\mathcal{I} \in [\![\mathtt{Proc}_{\setminus A, V} \bullet F]\!]\mathcal{M}$ such that the terminating model $\overline{\mathcal{M}}$ of $\mathcal{I}$ is also model of φ, i.e., $\overline{\mathcal{M}} \models \varphi$.*

Proof.

"$\Rightarrow$" We premise the satisfiability of (4.28) for all $i \in 1..n$ and consider a model $\mathcal{M} \models \psi$. We need to show that there is an interpretation $\mathcal{I}$ such that $\mathcal{I} \in [\![\mathtt{Proc}_{\setminus A, V} \bullet F]\!]\mathcal{M}$ with $\overline{\mathcal{M}} \models \varphi$ for the terminating model $\overline{\mathcal{M}}$.

Since $\psi \Rightarrow \psi_1 \vee \ldots \vee \psi_n$ there is a ψ_i with $\mathcal{M} \models \psi_i$ (if there are more than one matching ψ_i, we can choose one freely). Due to our premise, there has to be an interpretation $\mathcal{I}$ for this i that satisfies

$$(\lceil \psi \wedge \psi_i \rceil ^\frown \text{true}) \wedge F_i \wedge \bigwedge_{ev \in A} \boxminus ev \wedge (true ^\frown \updownarrow \checkmark ^\frown \lceil \varphi \rceil). \tag{4.29}$$

We assume that $\mathcal{I}(0)(e) = \mathcal{M}(e)$ for all events e, which can always be achieved, because in $\mathcal{I}$ the occurrence of events is only determined by *changes* of the event variables—$\mathcal{I}$ does not depend on the exact valuation of an event e at any point in time. Because of condition (4.27), there also is an interpretation

$\mathcal{I}' \in [\![F_i \wedge (true \frown \updownarrow\checkmark \frown \lceil\varphi\rceil)]\!]$ that equals $\mathcal{I}$, except for the initial model, which is replaced by $\mathcal{M}$. For the terminating model of $\mathcal{I}$, formula φ holds. Since all $v \in V$ are considered as DC constants (cf. proof of Thm. 4.5.3) and $\mathcal{I}' \models \bigwedge_{ev\in A} \boxminus\, ev$, we get the desired result

$$\mathcal{I}' \in [\![\texttt{Proc}_{\setminus A,V} \bullet F]\!]\mathcal{M}.$$

"$\Leftarrow$" We need to show the satisfiability of (4.28) for all ψ_i. Since $\psi \Leftrightarrow \psi_1 \vee \ldots \vee \psi_n$, there is a model $\mathcal{M}$ with $\mathcal{M} \models \psi$ and $\mathcal{M} \models \psi_i$. Using Remark 4.5.4, there is such an $\mathcal{M}$ that is not model of any other ψ_j with $j \neq i$, because otherwise ψ_i would be completely covered by the ψ_j. Therefore and due to definition of the ψ_i, $\mathcal{M}$ is initial model ($\mathcal{M} \in \{\mathcal{I} : [\![F_i]\!] \bullet \mathcal{I}(0)\}$) of F_i only. Using this model, we apply the precondition and get the interpretation

$$\mathcal{I} \in [\![\texttt{Proc}_{\setminus A,V} \bullet F]\!]\mathcal{M}$$

with terminating model $\overline{\mathcal{M}} \models \varphi$. Since $\mathcal{M}$ is only initial model of F_i, we also know $\mathcal{I} \in [\![F_i]\!]$. As the events in A and the global DC variables in V are both respected by $\texttt{Proc}_{\setminus A,V}$ and with $\mathcal{I}(0) = \mathcal{M} \models \psi \wedge \psi_i$, we can conclude the desired

$$\mathcal{I} \models (\lceil\psi \wedge \psi_i\rceil \frown \text{true}) \wedge F_i \wedge \bigwedge_{ev\in A} \boxminus\, ev \wedge (true \frown \updownarrow\checkmark \frown \lceil\varphi\rceil).$$

□

Finally, as for (diamond assumption$_\delta$) we can also check the side-condition of rule (assumption$_\diamond$) by checking whether

$$(\lceil\psi \wedge \psi_i\rceil \frown \text{true}) \wedge F_i \wedge \bigwedge_{ev\in A} \boxminus\, ev \wedge (\text{true} \frown \lceil\varphi\rceil \frown \text{true}) \text{ is satisfiable}$$

for all $i \in 1..n$ and the same preconditions as in (4.28). The correctness proof is almost identical to the proof of Thm. 4.5.5, but we do not need to consider the terminating model of interpretations.

Remark 4.5.6 (Over-approximation). Without an adequate set of configurations according to condition (4.27) it is still possible to check the following formulae

$$(\lceil\psi\rceil \frown \text{true}) \wedge F \wedge \bigwedge_{ev\in A} \boxminus\, ev \wedge (true \frown \updownarrow\checkmark \frown \lceil\neg\varphi\rceil) \text{ is } \textit{unsatisfiable}$$

$$(\lceil\psi\rceil \frown \text{true}) \wedge F \wedge \bigwedge_{ev\in A} \boxminus\, ev \wedge (true \frown \updownarrow\checkmark \frown \lceil\varphi\rceil) \text{ is satisfiable.}$$

This check is an over-approximation, because it is shown that *every* terminating model of an interpretation starting in a model of ψ satisfies φ. The second condition ensures that there actually is such an interpretation.

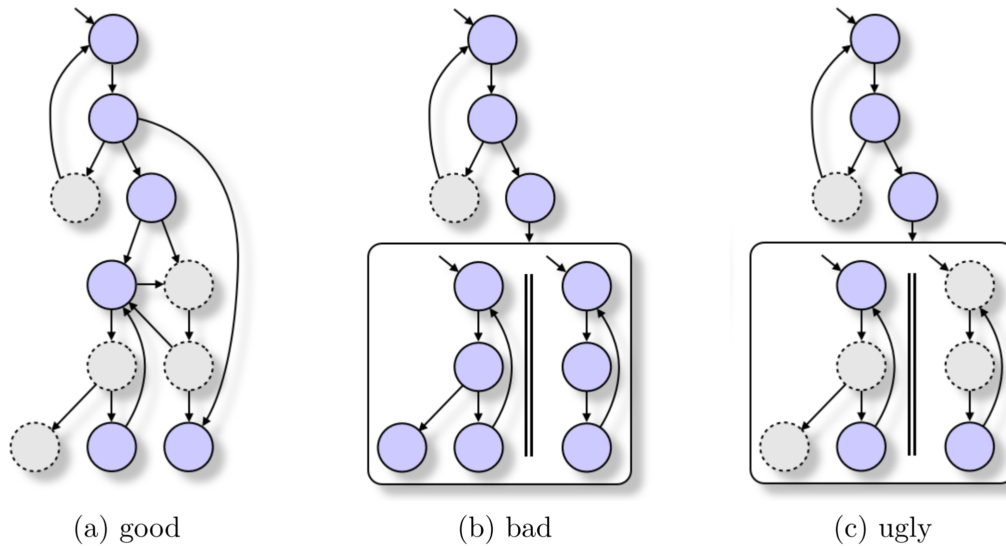


Figure 4.2.: Protocol structures to be handled with the DL approach

4.6. Discussion

4.6.1. Discussion of the VA Approach

One of the key ideas behind the VA approach is to provide a general formalism for specifying abstract behavioural patterns for real-time systems. In this chapter, we have introduced an approach to verify architectures with a sequent style calculus based on a Dynamic Logic (DL) extension for CSP processes. We have made use of DL as an instrument to compositionally reason about processes. For this reason, we benefit from the advantages of DL but also inherit its weaknesses.

Dynamic Logic was essentially developed by David Harel [Har79, Har84, HKT00] and there is a lot of work by several authors on using DL for automated verification of programs [HHRS86, HRS87, Bal05], with promising results with respect to tool support, which is one of the main reasons to choose DL as the basis for our proof calculus. For instance, there are the theorem provers *Karlsruhe Interactive Verifier (KIV)*[5] [HBB$^+$05] and *KeY*[6] [BHS07].

The use of DL to reason about VA processes works out particularly well on sequential protocol structures, that can be verified compositionally with the sequent calculus. Figure 4.2 sketches different shapes of protocol structures that we wish to handle with DL. In the best case (a), the considered process has a fully sequential structure with unknown parts and also recursions. Solid nodes in Fig. 4.2 represent normal sub-processes, whereas dashed nodes stand for constrained unknown pro-

[5] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/`
[6] `http://www.key-project.org/`

cesses. In the second situation (b), the structure also contains parallel parts over normal processes not containing unknown parts. These parallel parts can be handled with rule (parallel$_\Box$), that requires to unfold the process into a choice of processes. This is of course not ideal, because of the blow-up of the process structure by explicitly computing the parallel composition. This is a well-known issue of the DL approach that is, e.g., examined in [Bal05]. The most difficult situation we want to verify is sketched in (c): parallel processes over unknown parts with timing constraints over the unknowns and potentially shared variable access in the parallel components[7]. The verification of these process structures is difficult, because of the interaction of the parallel components due to timing constraints and because of the usual challenges with shared variable access [dRdBH$^+$01]. We have not given an answer on how to handle such processes in this chapter, but will discuss possible extensions of our calculus in Sect. 6.1 in order to cope with these issues. To handle such processes is particularly important, because unknown processes are the only processes that can be restricted with timed properties. Thus, restricting the overall timing behaviour of a VA process is only possible by a parallel composition with a constrained unknown process. Note that it is not possible to simply forbid shared variable access, because one of our main applications is the usage of VAs with combined specification formalisms that may allow shared variable access (for instance, CSP-OZ-DC allows shared variable access within classes).

4.6.2. Related Work

In the following, we compare our Dynamic Logic extension with similar approaches for CSP processes or timed languages. Afterwards, we review related work with respect to verification in the presence of unknown parts.

Dynamic logic for JCSP. In JCSP concurrency and communication for Java programs is modelled within a dedicated CSP Java library. The work [KRSS05] uses KeY [BHS07] to verify the Java part of JCSP programs and a translation to Petri nets for the CSP library calls. Recursion in CSP processes, infinite data, and timing constraints are not considered.

Dynamic temporal logic for hybrid programs. In [Pla07a, Pla08, Pla10] a dynamic logic extension for hybrid programs (dL) is introduced. A further extension of this work is the integration of temporal properties in [Pla07b]. The main difference to our work is that it focuses on hybrid programs instead of event-based communicating processes, and by this does not integrate as well with existing combined specification languages as our work. Unknown process parts that have shown to be useful in the context of design patterns for formal verification are not supported directly. Instead

[7] There may also be shared variable access in (b), which causes no problems as every possible mutual access is made explicit by unfolding the parallel composition.

Table 4.1.: Differences and similarities between dCSP and dL

	dL	dCSP
Specification of states	by formulae over $\mathbb{R}$	by many-sorted first-order formulae
System variables	system variables are non-rigid function symbols of arity 0	
Discrete updates	via value assignment	via first-order constraints
Programs	hybrid programs without communication	CSP processes with handshake communication
Parallel computing	parallel programs by preprocessing	parallel CSP processes
Control structure	regular program structures	context-free program structures
Unknown behaviour	star operator for undefined variables $x := *$	unknown processes `Proc`
Program extensions	by differential equations	by general (real-time) logics
Refinement	no notion of refinement	simple refinement rule for CSP-OZ-DC; easy to transfer to other combined formalisms

in dL (like in standard Dynamic Logic [HKT00]) one can explicitly set variables to undefined values using the star operator: $x := *$. With this it is possible to construct *unknown programs*, which are similar to unknown processes by expressions like

$$x := *;\ y := *;\ ?F(x, y).$$

Here, $F(x, y)$ is an expression in dynamic logic. This expression *could* encode large real-time formulae in hybrid programs, but this results generally in a large blow-up of the program size. For instance, in case of DC formulae the hybrid program has to reflect the power set automaton construction of Hoenicke [Hoe06, MFHR08], resulting in several, large parallel automata—each with a size exponential in the number of DC phases. Hence, it is not possible to directly (i.e., without further encoding) specify that real-time formulae have to be valid in the context of unknown parts of hybrid programs. A further advantage of using the standard language CSP is that we can apply existing techniques to reason about CSP—for instance, we could use a refinement checker for CSP (e.g., FDR[8] [Ros98]) to simplify a calculus proof for a complex CSP structure. In [PQ09] a fragment of the ETCS without concurrency

[8] `http://www.fsel.com/software.html`

is verified using dynamic logic for hybrid systems. In Sect. 8.2, we present an ETCS case study, solved with the VA approach, consisting of several communicating parallel components but without hybrid dynamics. Table 4.1 summarises differences and similarities between dL and dCSP.

Hybrid CSP calculus. The recent article [LLQ$^+$10] reports on ongoing work about a calculus for a hybrid extension of CSP (HCSP). It presents a logic based on Hoare Logic [Hoa69, AdBO09], DC, and Differential Invariants [PC08]. The used CSP dialect integrates continuous dynamics by allowing differential equations to occur in processes. It also incorporates a time-out operation similarly to Timed CSP [Sch99]. Parallel composition is restricted such that no data, neither continuous nor discrete, is shared between parallel components and no recursive processes are analysed. The work on the calculus is not completed yet; for instance, it has not been proven to be sound.

Verification in the presence of unknown components. The AVACS sub-project S1 develops techniques to compositionally verify complex systems in the presence of black box components in partial designs. Sven Schewe [Sch08] solved general decidability problems for the synthesis of black box implementations in partial designs: he showed that synthesis in partial designs is decidable if and only if the system contains no information fork. For the decidable case, the algorithm is exponential in the number of black boxes. Mealy automata with shared variables are used as model. A partial design with black box processes can also be seen as a pattern similar to Verification Architectures. Like in this work, the architecture has to be given manually.

In addition, in the context of the sub-project S1, Finkbeiner et al. [FSB06] examine automated synthesis of assumptions for single components of larger systems. In [FPS08, FPS10] this work is extended to timed systems modelled as timed automata.

While both approaches, the S1 approach and the VA approach, need an architecture given in advance, it is the goal of S1 alone to automatically partition the architecture into black and white boxes, to automatically derive assumptions for black boxes, and to automatically establish properties for partial designs. By contrast, the work presented here focuses on combined specification formalisms and provides a general framework for semi-automatically establishing properties for given VAs. These can be instantiated by given combined specifications—even if no automated assumption generation and partial design verification is possible. Due to the general undecidability [PR90] of the distributed synthesis problem, there will always be such cases where for a given architecture no automated synthesis is possible.

In contrast to our examination of combined specification languages, S1 considers basic formalisms like shared-variable Mealy processes. It is not directly clear how to use the results of S1 in the context of combined specification languages. Even though it is possible to translate combined specifications into its operational semantics, the parallel product of CSP, OZ, and DC part has to be computed, by which all struc-

tural information gets lost. It makes not much sense to consider the CSP, OZ, or DC parts as single components in a complex architecture, because of the tight integration between them. The VA approach can be seen as a step towards an integration of the S1 techniques with combined specification languages, because it answers the question of how to structure a combined specification into a partial design that may be used for compositional verification: that is, processes are used to define the architectures with data constraints, unknown processes as black boxes, and additional temporal assumptions over them; and we establish a refinement relation between so-defined architectures and a combined specification language. An interesting question for future work is to use S1 techniques to automatically generate architectures from the CSP and OZ parts of a given concrete specification, or to automatically generate assumptions on unknown components.

5 Refinement of Verification Architectures

After all, this is completely straightforward.
What could possibly go wrong?

(Dream, in Brief Lives, *Neil Gaiman)*

Any view of things that is not strange is false.

(A voice in the desert, in Fables & Reflections, *Neil Gaiman)*

5.1. Refinement of Verification Architectures 105
5.2. Simulation of Processes 109
5.3. Proof Rules for Checking Refinement 110
5.4. Syntactical Proof Rule for Process Refinement 114
5.5. Property Inheritance 119
5.6. Discussion 122

5.1. Refinement of Verification Architectures

In this section, we examine how CSP processes with data are refined. In particular, we consider the refinement not only by other CSP processes but also by parametric CSP-OZ-DC specifications. We provide a simple, sound rule to prove refinement by CSP-OZ-DC specifications syntactically. We recall that in our approach we verify instantiation relations in two steps: assumptions on unknown processes are verified by customary verification approaches for the temporal logic, and a refinement relation for the process structure needs to be established. Thus, we here provide a matching rule to prove refinement by CSP-OZ-DC specifications syntactically. To this end, the

assumptions on unknown processes are ignored, and a refinement check is performed on the structure of a VA.

In the remainder of this chapter, we first establish a refinement relation between VAs and CSP-OZ-DC specifications, and based on this relation, we introduce a proof rule for checking refinement. Finally, a matching rule is presented. We start by defining the notion refinement as for the trace semantics of CSP [Ros98]:

Definition 5.1.1 (Refinement of CSP processes)
Given CSP processes without data constraints P, Q we say that Q refines *P, written $P \sqsubseteq Q$, iff the set of traces of Q, $[\![Q]\!]$, is contained in the set of traces of P:*

$$[\![P]\!] \supseteq [\![Q]\!].$$

We extend this refinement relation to CSP processes with data constraints, and we define refinement of P by Q, written $P \sqsubseteq Q$, by

$$\forall \mathcal{M} : Model \bullet [\![P]\!]\mathcal{M} \supseteq [\![Q]\!]\mathcal{M}.$$

That is, P allows more behaviour than Q or Q is more defined than P.

We do not distinguish these two notions of refinement syntactically, but it should be clear from the context which variant is used.

We further extend this definition to refinement of VAs by CSP-OZ-DC specifications. CSP processes with data and CSP-OZ-DC have by construction the same semantical domain except that channels are declared differently in both specification languages as also discussed in [Hoe06]. On the side of CSP, channels have tuple type, whereas on the side of CSP-OZ-DC channels have schema type. But the channels can be easily transformed in the other representation:

Remark 5.1.2 (Transformation of channels). If a CSP channel $c : S_1 \times S_2$ is given, the corresponding CSP-OZ-DC channel depends on the choice of variable names in the channel declaration. For instance, a possible channel declaration is $c : [x_1 : S_1;\ x_2 : S_2]$. At level of CSP-OZ-DC the order of the channel parameters is not relevant, even though it is at CSP level. That is, both expressions[1]

$$c\langle\!|\ x_2{=}{=}v;\ x_1{=}{=}w\ |\!\rangle \quad \text{and} \quad c\langle\!|\ x_1{=}{=}w;\ x_2{=}{=}v\ |\!\rangle$$

represent the same, well-defined event belonging to the CSP-OZ-DC channel declaration of c. The equivalent representation at CSP level is $c.w.v$. To simplify the presentation, we will consider both representations as equivalent without explicitly stating the correlation between the channel declarations on CSP-OZ-DC and CSP level. We will particularly not distinguish the different representations in the following definitions of refinement.

[1] Hoenicke uses the Z binding presentation (cf. page 18 and [Hoe06, ISO02]) for compound channels: the == symbol within $\langle\!|\,|\!\rangle$-brackets binds channel parameter names to corresponding values, here v and w.

Firstly, we give a straightforward definition of refinement between CSP processes and CSP-OZ-DC and continue by a more sophisticated definition including refinements that consider different names and data types.

Definition 5.1.3 (Refinement of CSP processes with data by CSP-OZ-DC specifications)
Given a CSP process VP (VA process) with data and a CSP-OZ-DC specification cod, a refinement *of VP by cod, written VP* $\sqsubseteq$ *cod, is given iff*

$$\forall \mathcal{M}_{env} : Model \bullet$$
$$\{\mathcal{M}_{init} : Init(cod) \mid \mathcal{I} \in [\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init}) \bullet \mathcal{I}\} \quad \supseteq \quad [\![cod]\!]^{\mathcal{E}} \mathcal{M}_{env}, \tag{5.1}$$

where $Init(cod)$ *is the set of all models that are valid initial models of the specification cod (which may consist of several CSP-OZ-DC classes and defining paragraphs).*

This definition implies that the symbols, which are introduced in *cod* and thus are interpreted by the model $\mathcal{M}_{init}$, coincide with the symbols of the signature of a refined process *VP*. The definition needs to bridge the gap between the different semantical approaches of processes with data and CSP-OZ-DC specifications. Therefore, we need to override (using the standard Z operator $\oplus$) the environmental models $\mathcal{M}_{env}$ for symbols in *VP* and *cod* by models $\mathcal{M}_{init}$ for symbols that are introduced in *cod*, which are implicit in the semantics of CSP-OZ-DC but not in the semantics of processes with data.

Even though this definition does not explicitly impose restrictions on how symbols are declared and used in *cod*, it implicitly enforces the desired behaviour: whenever a symbol is changed in an execution of *VP* and occurs in a corresponding execution of the CSP-OZ-DC specification, the symbol must be declared in the state schema of *cod* – otherwise, a change of this symbol would not be possible in any execution of *cod*.

For the sake of applicability of our approach, we do not want to restrict ourselves to refinements of CSP processes where all symbols of the abstract architecture directly reoccur in the concrete instantiating CSP-OZ-DC specification. Instead, it is helpful to establish a relation between the symbols of the CSP process and the CSP-OZ-DC specification. By this we can also analyse instantiations of abstract data types by more concrete data types. Thus, we generalise the refinements of Def. 5.1.3 to refinements respecting an instantiation relation.

Definition 5.1.4 (Generalised refinement of CSP processes with data)
We consider a CSP process VP with data, a CSP-OZ-DC specification cod, and an instantiation schema $inst = [Decl(cod), SysVar, Const \mid \varphi]$, *where* $Decl(cod)$ *is the set of all declarations, e.g., from state schemas or declaring paragraphs, occurring in a CSP-OZ-DC specification. Let* $[\![inst]\!] \subseteq Interpretation$ *be the set of all interpretations respecting inst, i.e.,* $\forall\, t : Time, \mathcal{I} : [\![inst]\!] \bullet \mathcal{I}(t) \in [\![inst]\!]^{\mathcal{P}}$.

Then, a refinement *of VP by cod that* respects *inst, written* $VP \sqsubseteq_{inst} cod$*, is given iff*

$$\forall \mathcal{M}_{env} : Model \bullet$$
$$\{\mathcal{M}_{init} : Init(cod) \mid \mathcal{I} \in [\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init}) \bullet \mathcal{I}\} \quad \supseteq \quad [\![cod]\!]^{\mathcal{E}} \mathcal{M}_{env} \cap [\![inst]\!]. \tag{5.2}$$

The idea of inclusion (5.2)—that equals (5.1) except for the intersection on the right side of the inclusion—is to restrict the examined traces of *cod* to interpretations that respect *inst*, such that *inst* selects just the interpretations where the symbols of *cod* and *VP* are in the desired relation.

The schema *inst* represents an instantiation relation that maps symbols from the abstract CSP process with data to concrete realising symbols of the CSP-OZ-DC specification. For instance, if an abstract data type like a list over arbitrary objects is realised by a list of integer values, then the instantiation schema maps the abstract list to the concrete one.

Example 5.1.5 (Instantiation relation for the running example). The VA process for the running example from Sect. 3.1.4 contains the system variables *sf*, *ok* and constants *RD*, *CT*. However, in a concrete realisation of the architecture (see Sect. 8.1) the abstract safety value *sf* is usually replaced by concrete values for, e.g., positions of a train and a movement authority (MA). In the CSP-OZ-DC specification instantiating the VA process of the example, the system variables *ma* and *pos* are used to model the current MA position and the train position, respectively. Thus, the safety of the system can be described by the distance of the train to the end of the MA. Similarly, a constant *maxbd* contains the maximal braking distance for the train and *maxcd* the *check distance*, which is the maximal distance a train can move during one check cycle. The distance *RD* that is examined during the check cycle of the VA can be computed by summation of *maxcd* and *maxbd*.

Hence, the instantiation relation for the train control example is given by the following schema:

```
┌─ inst ─────────────────────────────
│ sf, RD : ℝ
│ pos, ma, maxbd, maxcd : ℝ
├──────────
│ sf = ma − pos
│ RD = maxbd + maxcd
└────────────────────────────────────
```

The remaining symbols, *ok* and *CT* are not mapped to different symbols and are used like specified in the VA.

5.2. Simulation of Processes

Since *simulation* is an established proof method to check refinement of processes [He89], we introduce the notion simulation below (following [Mil99]) to verify that a CSP-OZ-DC specification refines a CSP specification.

Definition 5.2.1 (Simulation on CSP processes)
A relation $\sim$: *Processes* $\leftrightarrow$ *Processes on processes is a* simulation *iff*

$$P \sim Q \text{ and } Q \stackrel{a}{\Longrightarrow} \overline{Q} \text{ implies}$$

$$\text{that there is a process } \overline{P} \text{ with } P \stackrel{a}{\Longrightarrow} \overline{P} \text{ and } \overline{P} \sim \overline{Q}.$$

A process P simulates *a process Q, written $P \preceq Q$, iff there exists such a relation $\sim$ from process P to Q. The event a may also be an internal event τ.*

Note that this definition actually defines a *forward-simulation* (also called down-simulation). But since it is the only simulation relation we need here we refer to it simply by the term *simulation*.

We prove a lemma about simulation, which we need to show the correctness of the refinement rules in Thm. 5.3.1 and 5.3.2. It constitutes the relationship between simulation and refinement of processes (without data constraints).

Lemma 5.2.2
Given two processes P and Q (without data) such that P simulates Q ($P \preceq Q$), we can conclude that Q is a refinement of P

$$P \sqsubseteq Q$$

in the trace semantics of CSP.

Proof. Since $P \preceq Q$, there is a relation $P \sim Q$ according to Def. 5.2.1. It is to be shown that $[\![P]\!] \supseteq [\![Q]\!]$. Let s be a trace of $[\![Q]\!]$ with a corresponding LTS run

$$Q \stackrel{s}{\Longrightarrow} \overline{Q};$$

we show by induction over the length of s that it is also a trace of $[\![P]\!]$ and that there is a corresponding LTS run

$$P \stackrel{s}{\Longrightarrow} \overline{P} \quad \text{with} \quad \overline{P} \sim \overline{Q}.$$

$s = \langle\rangle$: Then, s is trivially a trace of P. Additionally, by prerequisite, $P \sim Q$. Hence, with Def. 5.2.1 there is a process $\overline{P}$ with $P \stackrel{\langle\rangle}{\Longrightarrow} \overline{P}$ and $\overline{P} \sim \overline{Q}$.

$s = w \frown \langle a\rangle$: The event sequence s is a trace of $[\![Q]\!]$ with an LTS run

$$Q \stackrel{w}{\Longrightarrow} \overline{Q} \stackrel{a}{\Longrightarrow} R.$$

By applying the induction hypothesis to the sub-sequence w, w is a also a trace of $[\![P]\!]$, i.e., the LTS of P has a run

$$P \stackrel{w}{\Longrightarrow} \overline{P} \quad \text{with} \quad \overline{P} \sim \overline{Q}.$$

Now, as $\overline{P} \sim \overline{Q}$ is a simulation according to Def. 5.2.1, and due to $\overline{Q} \stackrel{a}{\Longrightarrow} R$, there exists a process S with $\overline{P} \stackrel{a}{\Longrightarrow} S$ and $S \sim R$. Moreover, with the run

$$P \stackrel{w}{\Longrightarrow} \overline{P} \stackrel{a}{\Longrightarrow} S$$

of the LTS of P the trace $s = w \frown \langle a \rangle$ is also in $[\![P]\!]$.

□

This lemma is a standard result in the theory of CSP. For the stable-failures semantics and the failures-divergences semantics this result holds in a similar way.

5.3. Proof Rules for Checking Refinement

With the following proof rules, a refinement relation between a CSP-OZ-DC specification and a CSP process can be verified. It is based on a simulation relation between the `main` process and the unconstrained CSP process. In contrast to that, the definition of *matching* in Def. 5.4.1 gives rise to a purely syntactical proof rule.

Theorem 5.3.1 (Refinement rule)
Let VP be a CSP process with data, and let cod be a CSP-OZ-DC class with CSP process `main`*. Then, cod refines VP, $VP \sqsubseteq cod$, if the following conditions hold:*

1. a) *The process VP with all data constraints removed (using the unconst function defined in Sect. 3.1.1) simulates the process* `main`*:*

   $$unconst(VP) \preceq \texttt{main}.$$

   b) *For every sub-process P of* `main` *that is in 1a) simulated by an unknown process $\texttt{Proc}^{(\infty)}_{\backslash A,V}$ we demand that the system variables from V are not changed, i.e., given an operation schema* `com_a` *for $a \in alph(P)$ no changeable system variable from the delta list $\Delta(s_1, \ldots, s_n)$ is in V, so $s_i \notin V$ for $i \in 1..n$.*

2. *The symbols of the signature Σ of VP coincide with the symbols introduced in cod. That is, for $\Sigma = (Sort, SysVar, Const, Var)$ the types of cod correspond to the sorts Sort, state variables of cod to symbols from SysVar, global constants to Const, and message variables to variables from Var.*

3. *For all occurrences of an event* $a \bullet \varphi$ *in VP, declared by* $a : [x_1 : S_1; \ldots; x_n : S_n]$*, system variables* $s_1, \ldots, s_m$ *and primed system variables* $u'_1, \ldots, u'_l$ *occurring in* φ*, there is an operation schema*

$$\texttt{com_}a = [\Delta(u_1, \ldots, u_l);\ x_1 : S_1;\ \ldots;\ x_n : S_n \mid \varphi]$$

in cod, and the function symbols are declared in the state schema: $s_i \in State(cod)$ *for* $i \in 1..n$*. The variables* x_1 *to* x_n *are the message variables of channel a.*

The last condition also implies that for two events $a \bullet \varphi_1$ and $a \bullet \varphi_2$ from the process *VP* the condition $\varphi_1 = \varphi_2$ is true, because different definitions of $\texttt{com_}a$ are not allowed in CSP-OZ-DC specifications.

The refinement rule 5.3.1 is defined with respect to the simplified refinement notion from Def. 5.1.3, demanding that symbols from CSP-OZ-DC specification and VA process have to coincide. In the case of the generalised refinement of Def. 5.1.4, an instantiation relation *inst* is used to establish more sophisticated connections. The following rule extends rule 5.3.1 by incorporating such an instantiation relation.

Theorem 5.3.2 (Generalised refinement rule)
With the same premises as in 5.3.1, cod is a refinement of VP respecting the instantiation schema $inst = [Decl(cod), SysVar, Const \mid \psi]$*, i.e.,*

$$VP \sqsubseteq_{inst} cod,$$

if conditions 1. and 2. from Thm. 5.3.1 hold (under consideration of inst) and

3. *For every constrained occurrence* $a \bullet \varphi$ *in VP, declared by* $a : [x_1 : S_1; \ldots; x_n : S_n]$*, system variables* $s_1, \ldots, s_m$ *and primed system variables* $u'_1, \ldots, u'_l$ *occurring in* φ*, there is an operation schema*

$$\texttt{com_}a = [\Delta(u_1, \ldots, u_l);\ x_1 : S_1;\ \ldots;\ x_n : S_n \mid \overline{\varphi}]$$

in cod, such that

$$(\psi \wedge \psi' \wedge \overline{\varphi}) \Rightarrow \varphi$$

holds, that is, $\overline{\varphi}$ *implies the constraint* φ *of the CSP process, as long as the instantiation constraint* ψ *is respected.*

In this rule, we allow that the refinement system *cod* introduces new deadlocks, which is the case if $\overline{\varphi}$ is stronger than φ such that the execution of *a* is blocked in the refinement *cod* but not in the process *VP*. This difference to [Fis00] is possible, since we do not consider failure semantics here.

Remark 5.3.3 (Uniqueness of *inst*). The generalised refinement rule can only be applied if the instantiation relation *inst* uniquely assigns valuations of *VP* symbols to *cod* symbols. This is a consequence of condition 1b), which demands that a process

refining an unknown process $\texttt{Proc}^{(\infty)}_{\backslash A,V}$ does not change variables in V. This can only be achieved in the case that the symbols of the refining process are uniquely mapped to symbols in V – otherwise the valuation of the V symbol could be changed with every event, because multiple values are allowed by *inst*. For instance, in the relation of Example 5.1.5 a constraint like $sf < ma - pos$ would not be possible: with such an instantiation relation the symbol *sf* can be changed in every operation. To avoid suchlike behaviour, we only allow unique instantiation relations.

Correctness of refinement rule. The refinement rule of Thm. 5.3.1 is correct, i.e., the conditions *1.* up to *3.* actually imply the refinement

$$VP \sqsubseteq cod.$$

Proof. To prove $VP \sqsubseteq cod$ we need to show that the inclusion (5.1) of Def. 5.1.3 holds. To this end, we consider an arbitrary model $\mathcal{M}_{env} : Model$ and an interpretation

$$\mathcal{I} : [\![cod]\!]^{\mathcal{E}} \mathcal{M}_{env} \tag{5.3}$$

with an untimed sequence $\langle \mathcal{M}_0, \mathcal{M}_1, \ldots \rangle$. The proof is finished if we can show that there is a model $\mathcal{M}_{init}$ with

$$\mathcal{I} \in [\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init}). \tag{5.4}$$

This is the case if the semantical conditions from Sect. 3.1.1 on page 46 are true. We provide evidence that these conditions hold:

1. There is a run π of the LTS of $unconst(VP)$ such that π fits to $\mathcal{I}$. This holds, because we know from (5.3) and the semantics of *cod* that $\mathcal{I} \models \texttt{main}$, that is, there is a run π of the LTS of $\texttt{main}$: $\pi \in [\![\texttt{main}]\!]$. From condition *1a)* of rule 5.3.1 and Lemma 5.2.2, it can be inferred that

$$unconst(VP) \sqsubseteq \texttt{main}.$$

   Hence, π, for that we already know that it fits to $\mathcal{I}$, is also a run of $unconst(VP)$.

2. We need to show that there is an initial model $\mathcal{M}_{init} : Init(cod)$ of *cod* such that the first model $\mathcal{M}_0$ of the interpretation $\mathcal{I}$ is equal to $\mathcal{M}_{env} \oplus \mathcal{M}_{init}$. This is true by construction of $Init(cod)$, which contains all possible initial models of *cod* respecting the three parts CSP, OZ, and DC, and the local paragraphs. Since the semantics of *cod*, given by $[\![cod]\!]^{\mathcal{E}} \mathcal{M}_{env}$, is not empty, there actually is such a model, namely $\mathcal{M}_{init}$, that possibly overwrites symbols defined in $\mathcal{M}_{env}$. Altogether, the interpretations of the *cod* semantics need to respect the environment model $\mathcal{M}_{env}$ and initially the model $\mathcal{M}_{init}$ from $Init(cod)$ (which covers that models from local paragraphs may overwrite symbols from the environment). Thus,

$$\mathcal{M}_0 = \mathcal{M}_{env} \oplus \mathcal{M}_{init}.$$

3. We examine the untimed sequence $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$ of $\mathcal{I}$: Let $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ be models with $\mathcal{M}_{i-1}(a) \neq \mathcal{M}_i(a)$, i.e., the event a induces a state change from $\mathcal{M}_{i-1}$ to $\mathcal{M}_i$. We consider two cases:

   a) $a \bullet \varphi$ occurs in the process *VP* directly (i.e., not only as possible event of an unknown process): condition *3.* from Thm. 5.3.1 yields that there is a communication schema $\texttt{com_}a = [S \mid \varphi]$, where S is a declaration. Due to (5.3) and the CSP-OZ-DC semantics we can conclude $\mathcal{M}_{i-1} \cup \mathcal{M}'_i \in [\![\texttt{com_}a]\!]^{\mathcal{P}}$ and by this $\mathcal{M}_{i-1} \cup \mathcal{M}'_i \models \varphi$. Since the symbols from *cod* and the signature of *VP* coincide as demanded in condition *2.* of Thm. 5.3.1, we also get

   $$(\mathcal{M}_{i-1} \cup \mathcal{M}'_i) \models \varphi$$

   in the signature of *VP*.

   b) a does not explicitly occur in *VP*, i.e., the event a occurs within an unknown process $\texttt{Proc}^{(\infty)}_{\backslash A, V}$. Hence, according to the semantics of unknown processes, the occurrence of a in the LTS of *VP* is constrained by $a \bullet \Xi V$, and we need to show

   $$\mathcal{M}_{i-1} \cup \mathcal{M}'_i \models \Xi V \tag{5.5}$$

   Due to condition 1b) of Thm. 5.3.1, if there is an operation schema $\texttt{com_}a$ in a process refining $\texttt{Proc}^{(\infty)}_{\backslash A, V}$ with $\Delta(s_1, \dots, s_n)$, then $s_i \notin V$ (hence, no symbol from V can be changed). Thus, (5.5) actually holds.

4. To show the fourth semantical condition from page 46, we consider an arbitrary parameter $v \in \mathit{Const}$. Due to condition 2. of Thm. 5.3.1, it corresponds to a constant declaration of *cod*. Since global constants cannot be changed, the CSP-OZ-DC semantics directly yields the desired condition $\mathcal{M}_{i-1}(v) = \mathcal{M}'_i(v)$.

Thus, all semantical conditions are fulfilled, and we can actually conclude (5.4), i.e., that $\mathcal{I}$ is an interpretation of *VP* for $\mathcal{M}_{env} \oplus \mathcal{M}_{init}$. Since we chose $\mathcal{I}$ arbitrarily, inclusion (5.1) holds and, by this, $\mathit{VP} \sqsubseteq \mathit{cod}$. □

Correctness of generalised refinement rule. The generalised refinement rule of Thm. 5.3.2 is correct.

Proof. The proof is very similar to the proof of Thm. 5.3.1, particularly, the proof steps 1. and 2. coincide. The only difference is that we need to consider the modified condition 3. of Thm. 5.3.2 and the refinement relation $\mathit{inst} = [\mathit{Decl}(\mathit{cod}), \mathit{SysVar}, \mathit{Const} \mid \psi]$ for all state changes in the examined interpretation $\mathcal{I}$. As we need to show inclusion (5.2), we only consider interpretations $\mathcal{I}$ that respect *inst*, so $\mathcal{I} : [\![\mathit{cod}]\!]^{\mathcal{E}} \mathcal{M}_{env} \cap [\![\mathit{inst}]\!]$.

We examine an event a of *cod* with $\texttt{com_}a = [S \mid \overline{\varphi}]$ and a state change in $\mathcal{I}$ triggered by this event a, $\mathcal{M}_{i-1} \cup \mathcal{M}'_i \models \overline{\varphi}$. Since $\mathcal{I}$ is an interpretation respecting

inst, $\mathcal{M}_{i-1} \cup \mathcal{M}'_i \models \psi \wedge \psi'$. For an a occurring directly in VP, we can apply condition 3. of 5.3.2 and get that there is a corresponding event $a \bullet \varphi$ of VP and

$$(\psi \wedge \psi' \wedge \overline{\varphi}) \Rightarrow \varphi.$$

Therefore, $\mathcal{M}_{i-1} \cup \mathcal{M}'_i \models \varphi$. If a only occurs in sub-processes refining an unknown process, then with 1b) and the instantiation relation *inst* we get that variables restricted by the unknown process cannot be changed (see also Remark 5.3.3).

Since this argument holds for all state changes of $\mathcal{I}$ we can conclude

$$\mathcal{I} \in [\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init})$$

and, by this, inclusion (5.2). □

5.4. Syntactical Proof Rule for Process Refinement

We now give a definition of a *matching* between processes and CSP-OZ-DC specifications. This definition can be used as a proof rule that establishes a refinement relation between a CSP process with data and a CSP-OZ-DC specification. The rule is not complete, i.e., not all valid refinements can be shown applying the rule. But this is not our goal here, because in our application scenario concrete realisations are modelled with respect to a given VA, and thus, we assume that the concrete model reflects the structure of the VA directly. Hence, our proof rule connects abstract processes with concrete models of a particular well-fitting shape.

Definition 5.4.1 (Matching)
Let VP be a CSP process with data in `Proc` *normal form with characteristic process CP and unknown processes $U_1 \stackrel{c}{=} \mathtt{Proc}^{(\infty)}_{\backslash A_1, V_1}, \ldots, U_n \stackrel{c}{=} \mathtt{Proc}^{(\infty)}_{\backslash A_n, V_n}$, and let cod be a CSP-OZ-DC class with CSP process* `main`. *We say that cod* matches *VP if conditions 2. and 3. from Thm. 5.3.1 or from Thm. 5.3.2 are valid, and instead of condition 1. the following syntactical conditions hold:*

1. a) `main` $= \overline{CP}[Y_1/U_1, \ldots, Y_n/U_n]$, *where $\overline{CP} = unconst(CP)$ and the Y_i are new process identifiers defined as $Y_i \stackrel{c}{=} CP_i$ for $i \in 1..n$. That is, the* `main` *process structurally equals VP, except that all unknown processes are replaced by implementing processes. We additionally demand that processes implementing* $\mathtt{Proc}^{\infty}$ *do not contain the* `Skip` *process, i.e., they do not terminate.*

   b) *For every process CP_i we demand that (1) the forbidden events from A are respected, $alph(CP_i) \cap A_i = \varnothing$, and (2) the function symbols from V are not changed, i.e., given an operation schema* `com_`*a for $a \in alph(CP_i)$ no changeable function symbol from the delta list $\Delta(s_1, \ldots, s_n)$ is in V, so $s_i \notin V$ for $i \in 1..n$.*

Instead of checking the simulation relation on the processes directly, as it is done in condition *1.* of 5.3.1 and 5.3.2, a matching demands that the `main` process of a CSP-OZ-DC specification syntactically coincides with the process *VP* except for the unknown processes that are implemented by concrete processes in `main`.

Example 5.4.2 (Matching processes). As an example, let us consider a part of the VA process of the running example:

$$\begin{aligned} System &\overset{c}{=} \big(FAR \,\S\, check \bullet \varphi_{check} \\ &\qquad \rightarrow (fail \bullet \varphi_{fail} \rightarrow REC \,\Box\, pass \bullet \varphi_{pass} \rightarrow System)\big) \\ FAR &\overset{c}{=} \mathtt{Proc}_{\setminus A,C} \bullet F_{FAR} \\ REC &\overset{c}{=} \mathtt{Proc}^{\infty}_{\setminus A,C} \bullet F_{REC}. \end{aligned}$$

This process already is in `Proc` normal form, because it consists of the characteristic *System* process and only has references to sub-processes *FAR* and *REC*, representing unknown processes.

The following instantiating `main` process matches *System* according to Def. 5.4.1:

$$\begin{aligned} \mathtt{main} &\overset{c}{=} \big(\overline{FAR} \,\S\, check \\ &\qquad \rightarrow (fail \rightarrow \overline{REC} \,\Box\, pass \rightarrow \mathtt{main})\big) \\ \overline{FAR} &\overset{c}{=} P_1 ||| P_2 ||| P_3 \\ \overline{REC} &\overset{c}{=} applyEB \rightarrow (\mu\, X \bullet updSpd \rightarrow updPos \rightarrow X), \end{aligned}$$

where P_1 up to P_3 are further concrete sub-processes (see Sect.8.1 for details). These example shows that the processes `main` and *System* coincide syntactically except that the event constraints do not occur in `main` and except that process references to unknown processes are exchanged by new references to concrete refining processes $\overline{FAR}$ and $\overline{REC}$, respecting the exclusion alphabet A of the unknown processes.

The next theorem shows that condition *1.* of Def. 5.4.1 actually implies that the process *VP* simulates the `main` process. By this means, matching can be used as a syntactical proof rule to verify process refinement by CSP-OZ-DC specifications.

Theorem 5.4.3 (Matching implies simulation)
Given a CSP-OZ-DC specification cod and a process VP in `Proc` *normal form that respect condition 1. from the definition of matching in Def. 5.4.1. We can conclude that*

$$unconst(VP) \preceq \mathtt{main}.$$

Proof. We start by defining the relation that establishes the simulation between *VP* and *cod*. Since `main` equals the unconstrained characteristic process $\overline{CP}$ except for

processes implementing unknown parts, which is in Def. 5.4.1 specified with the renaming $\overline{CP}[Y_1/U_1, \ldots, Y_n/U_n]$ for $Y_i \stackrel{c}{=} CP_i$, the desired simulation is defined as follows:

$$\begin{aligned} \sim := &\{(R, R[Rn]) \mid R \neq \texttt{Proc} \wedge R \neq P_1 \, op \, P_2\} \cup \\ &\{(P_1 \, op \, P_2, Q_1 \, op \, Q_2) \mid P_1 \sim Q_1 \wedge P_2 \sim Q_2\} \cup \\ &\{(\texttt{Proc}^{(\infty)}_{\backslash A_i, V_i}, \overline{CP}_i) \mid CP_i \Longrightarrow^* \overline{CP}_i\}, \end{aligned} \tag{5.6}$$

where $[Rn]$ represents the renaming $[Y_1/U_1, \ldots, Y_n/U_n]$. The first case $(R, R[Rn])$ applies to all non-binary processes except Proc, e.g., Skip and Stop or prefixing processes and application of recursion. The second case applies to all binary CSP operators.

We show that this relation is indeed a simulation relation from *unconst*(*VP*) to main by proving the simulation condition of Def. 5.2.1 for CP and $\overline{CP}[Rn]$. Assuming that $Q \stackrel{a}{\Longrightarrow} \overline{Q}$ with $P \sim Q$, we have to show that there is a process $\overline{P}$ such that $P \stackrel{a}{\Longrightarrow} \overline{P}$ and $\overline{P} \sim \overline{Q}$. The proof is by induction over the structure of P:

1. Base cases:

   $P = \texttt{Stop}$: In this case also $Q = \texttt{Stop}$. Thus, no transition step is possible for P and also for Q.

   $P = \texttt{Skip}$:

   $$\begin{aligned} &P \sim Q && \stackrel{\{\text{Def. } \sim\}}{\Rightarrow} && Q = \texttt{Skip} \\ &Q \stackrel{a}{\Longrightarrow} \overline{Q} && \stackrel{\{\text{LTS}\}}{\Rightarrow} && a = \checkmark \text{ and } \overline{Q} = \Omega \end{aligned}$$

   The following transitions for P are possible:

   $$P \stackrel{\checkmark}{\Longrightarrow} \Omega$$

   With $\Omega \sim \Omega[Rn] = \Omega$ we get $\overline{P} \sim \overline{Q}$.

   $P = a \rightarrow R$:

   $$\begin{aligned} &P \sim Q && \stackrel{\{\text{Def. } \sim\}}{\Rightarrow} && Q = (a \rightarrow R)[Rn] = a \rightarrow R[Rn] \text{ and } R \sim R[Rn] \\ &Q \stackrel{a}{\Longrightarrow} \overline{Q} && \stackrel{\{\text{LTS}\}}{\Rightarrow} && R[Rn] = \overline{Q} \text{ or } R[Rn] \stackrel{\tau}{\Longrightarrow} \overline{Q} \end{aligned}$$

   Note that here R cannot be a Proc process due to the Proc normal form such that we actually can assume $R \sim R[Rn]$ according to (5.6).

   The following transitions for P are possible:

   $$P \stackrel{a}{\Longrightarrow} R$$

   One can show by induction over the structure of process expressions that whenever $R_1 \sim S_1$ and $S_1 \stackrel{\tau}{\Longrightarrow} S_2$, then there is also an R_2 with $R_1 \stackrel{\tau}{\Longrightarrow} R_2$ and

$R_2 \sim S_2$. In the concrete case, we apply this to $R \sim R[Rn]$ and $R[Rn] \stackrel{\tau}{\Longrightarrow} \overline{Q}$ and get that there is an $\overline{P}$ with $R \stackrel{\tau}{\Longrightarrow} \overline{P}$ and $\overline{P} \sim \overline{Q}$. Overall, this means that there is a transition $P \stackrel{a}{\Longrightarrow} \overline{P}$ with $\overline{P} \sim \overline{Q}$.

$\underline{P = \texttt{Proc}_{\backslash A_i, V_i} :}$

$$\begin{array}{lcl} P \sim Q & \stackrel{\{\text{Def. } \sim\}}{\Rightarrow} & Q = S \text{ with } CP_i \Longrightarrow^* S \\ Q = S \stackrel{a}{\Longrightarrow} \overline{S} & \stackrel{\{\text{LTS}\}}{\Rightarrow} & a \neq \checkmark \wedge CP_i \Longrightarrow^* \overline{S} \text{ or} \\ & & a = \checkmark \wedge \overline{Q} = \Omega \end{array}$$

If $a \neq \checkmark$, then $a \notin A_i$ because of condition *1b)* of Def. 5.4.1. Hence, the following transitions for P are possible:

$$P \stackrel{a}{\Longrightarrow} \texttt{Proc}^{(\infty)}_{\backslash A_i, V_i} = \overline{P}$$

With $CP_i \Longrightarrow^* \overline{S}$ we get $\overline{P} \sim \overline{S}$ (definition of $\sim$ for case $P = \texttt{Proc}$). For $a = \checkmark$ our claim holds due to $P \stackrel{\checkmark}{\Longrightarrow} \Omega$ (definition of `Proc`) and $\Omega \sim \Omega$.

$\underline{P = \texttt{Proc}^{\infty}_{\backslash A_i, V_i}}$: This case can be solved completely analogously to `Proc` except that we omit the case $a = \checkmark$.

$\underline{P = U_i :}$

$$\begin{array}{lcl} P \sim Q & \stackrel{\{\text{Def. } \sim\}}{\Rightarrow} & Q = Y_i, \text{ where } Y_i \stackrel{c}{=} CP_i \\ Q \stackrel{\tau}{\longrightarrow} CP_i \stackrel{a}{\Longrightarrow} \overline{Q} & \stackrel{\{\text{LTS}\}}{\Rightarrow} & a \neq \checkmark \wedge CP_i \Longrightarrow^* \overline{Q} \text{ or} \\ & & a = \checkmark \wedge \overline{Q} = \Omega \end{array}$$

The following transitions for P are possible:

$$\begin{array}{lcl} P \stackrel{\tau}{\longrightarrow} \texttt{Proc}^{(\infty)}_{\backslash A_i, V_i} \stackrel{a}{\Longrightarrow} \overline{P} & \Rightarrow & a \neq \checkmark \wedge \overline{P} = \texttt{Proc}^{(\infty)}_{\backslash A_i, V_i} \text{ or} \\ & & a = \checkmark \wedge \overline{P} = \Omega \end{array}$$

In both cases, $a \neq \checkmark$ and $a = \checkmark$, we directly get $\overline{P} \sim \overline{Q}$ from (5.6).

2. Compound cases:

   $\underline{P = \mu\, Z \bullet F(Z) :}$

$$\begin{array}{lcl} P \sim Q & \stackrel{\{\text{Def. } \sim\}}{\Rightarrow} & Q = \mu\, Z \bullet F(Z)[Rn] \text{ and } F(Z) \sim F(Z)[Rn] \\ Q \stackrel{a}{\Longrightarrow} \overline{Q} & \stackrel{\{\text{LTS}\}}{\Rightarrow} & Q \stackrel{\tau}{\longrightarrow} F(\mu\, Z \bullet F(Z)[Rn])[Rn] = F(P[Rn])[Rn] \\ & & = F(P)[Rn] \wedge F(P)[Rn] \stackrel{a}{\Longrightarrow} \overline{Q} \end{array} \tag{5.7}$$

   Implication (5.7) is obtained by application of the rule (recursion) of the operational CSP semantics. Note that we make additionally use of the equality $F(P[Rn])[Rn] = F(P)[Rn]$, which holds for arbitrary processes F and renamings Rn. To $F(P) \sim F(P)[Rn]$ and $F(P)[Rn] \stackrel{a}{\Longrightarrow} \overline{Q}$ we can apply the induction hypothesis yielding that there is a process $\overline{P}$ and a transition such that the following transition is possible:

$$F(P) \stackrel{a}{\Longrightarrow} \overline{P} \text{ with } \overline{P} \sim \overline{Q}$$

By this: $P \xrightarrow{\tau} F(\mu\, Z \bullet F(Z)) = F(P) \overset{a}{\Longrightarrow} \overline{P}$ with $\overline{P} \sim \overline{Q}$ as required.

<u>$P = P_1\, op\, P_2$:</u>

$$P \sim Q \quad \overset{\{\text{Def. } \sim\}}{\Rightarrow} \quad Q = Q_1\, op\, Q_2 \text{ with } P_1 \sim Q_1 \wedge P_2 \sim Q_2$$

We assume

$$Q \overset{a}{\Longrightarrow} \overline{Q}. \tag{5.8}$$

To examine how P may simulate this step, we need to distinguish the operators *op*:

a) $op = \fatsemi$: If $Q_1 = \Omega$, then the only possible transition is $Q \xrightarrow{\tau} Q_2$. Due to the semantics of $\fatsemi$ and requirement (5.8) there is a transition $Q_2 \overset{a}{\Longrightarrow} \overline{Q}_2 = \overline{Q}$. With the induction hypothesis, we get that there also is a transition $P_2 \overset{a}{\Longrightarrow} \overline{P}_2$ and $\overline{P}_2 \sim \overline{Q}_2$.

   Because of $\Omega = Q_1 \sim P_1 = \Omega$, we can simulate the transitions on the level of P:

$$P = \Omega \fatsemi P_2 \quad \text{and} \quad P \xrightarrow{\tau} P_2 \overset{a}{\Longrightarrow} \overline{P}_2$$

   with $\overline{P}_2 \sim \overline{Q}_2 = \overline{Q}$.

   If there is a transition $Q_1 \overset{a}{\Longrightarrow} \overline{Q}_1$ instead, we argue analogously: due to the induction hypothesis, there is a transition $P_1 \overset{a}{\Longrightarrow} \overline{P}_1$ with $\overline{P}_1 \sim \overline{Q}_1$. Hence, with a process $\overline{P} = \overline{P}_1 \fatsemi P_2$ there is a transition

$$P \overset{a}{\Longrightarrow} \overline{P} \text{ with } \overline{P}_1 \sim \overline{Q}_1 \wedge P_2 \sim Q_2.$$

   Since $\overline{P} = (\overline{P}_1 \fatsemi P_2)$, $\overline{Q} = (\overline{Q}_1 \fatsemi Q_2)$, and $\overline{P}_1 \sim \overline{Q}_1$ as well as $P_2 \sim Q_2$, we apply (5.6) in order to get $\overline{P} \sim \overline{Q}$.

b) $op = \Box$: due to the semantics of $\Box$ and requisite (5.8), there is a transition $Q_i \overset{a}{\Longrightarrow} \overline{Q_i}$ for $i \in 1..2$ with $\overline{Q} = \overline{Q}_i$. We apply the induction hypothesis with the result that there is a transition $P_i \overset{a}{\Longrightarrow} \overline{P}_i$ with $\overline{P}_i \sim \overline{Q}_i$. It follows that there is a transition $P \overset{a}{\Longrightarrow} \overline{P}_i$ likewise, which finishes this case.

c) $op = \sqcap$: Like $\Box$.

d) $op = \|_A$: There are three possibilities for transitions:

$$Q_1 \overset{a}{\Longrightarrow} \overline{Q}_1 \wedge Q_2 \overset{a}{\Longrightarrow} \overline{Q}_2 \qquad \text{for } a \in A$$

$$Q_i \overset{b}{\Longrightarrow} \overline{Q}_i \qquad \text{for } i \in 1..2, b \notin A.$$

   We start with the synchronisation case, in which $\overline{Q} = \overline{Q}_1 \|_A \overline{Q}_2$. We apply the induction hypothesis twice: there are transitions

$$P_1 \overset{a}{\Longrightarrow} \overline{P}_1 \wedge P_2 \overset{a}{\Longrightarrow} \overline{P}_2 \text{ with } \overline{P}_1 \sim \overline{Q}_1 \wedge \overline{P}_2 \sim \overline{Q}_2.$$

   By this, there is a transition $P \overset{a}{\Longrightarrow} \overline{P}$, where we set $\overline{P} = \overline{P}_1 \|_A \overline{P}_2$, and $\overline{P} \sim \overline{Q}$ holds.

In the second case, $\overline{Q} = \overline{Q_1} \parallel_A Q_2$, we have a transition $Q \overset{b}{\Longrightarrow} \overline{Q}$. We apply the induction hypothesis to $\overline{Q_1}$, get a transition $P_1 \overset{b}{\Longrightarrow} \overline{P}_1$ with $\overline{P}_1 \sim \overline{Q}_1$, and conclude that there is a transition step $P \overset{b}{\Longrightarrow} \overline{P}_1 \parallel_A P_2$, where $(\overline{P}_1 \parallel_A P_2) \sim (\overline{Q}_1 \parallel_A Q_2) = \overline{Q}$ due to the definition of $\sim$ in (5.6). The third case is completely symmetric to the second.

e) $op = \; |||$: This is a special case of $op = \parallel_A$.

We have proven now that $\sim$ is indeed a simulation relation, hence

$$unconst(VP) \preceq \texttt{main}.$$

□

The following corollary establishes our argument that matching induces a syntactic proof rule for the refinement relation between a CSP-OZ-DC specification and a process.

Corollary 5.4.4 (Matching implies refinement)
If a CSP-OZ-DC specification cod matches a process VP according to Def. 5.4.1, Thm. 5.4.3 yields

$$unconst(VP) \preceq \texttt{main}.$$

With this we can apply the refinement rule 5.3.1 or the generalised refinement rule 5.3.2 and get

$$VP \sqsubseteq cod \quad or \quad VP \sqsubseteq_{inst} cod,$$

respectively.

5.5. Property Inheritance

The aim of introducing a refinement notion in this chapter has been to ensure property inheritance from established safety properties of an extended CSP process to a CSP-OZ-DC specification. We recall that refinement has been defined in Def. 5.1.3 by

$$\forall \mathcal{M}_{env} : Model \bullet$$
$$\{\mathcal{M}_{init} : Init(cod) \mid \mathcal{I} \in [\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init}) \bullet \mathcal{I}\} \quad \supseteq \quad [\![cod]\!]^{\mathcal{E}} \mathcal{M}_{env}. \qquad (5.1)$$

Basically, if a safety property has been shown for all interpretations of an extended CSP process *VP* and a refinement between *VP* and a CSP-OZ-DC specification *cod* has been established, then one can directly conclude from inclusion (5.1) that this safety property also holds for *cod* (and similarly for the extended refinement notion respecting an instantiation relation).

Initial Models

Usually a property of an extended CSP process will not be established for all initial models but instead for a given set of initial models. For instance, if a property *safe* has been proven for an initial constraint φ_{init} by showing

$$\varphi_{init} \vdash [VP]safe,$$

then the safety property only holds for all interpretations of the refining specification *cod* that start with a φ_{init} model.

When examining the refinement rules of Sect. 5.3, we find that the `Init` schema of the CSP-OZ-DC specification is nowhere considered. The reason is that the initial constraints of the specification are implicitly considered in inclusion (5.1): only interpretations are taken into account that start with models $\mathcal{M}_{init}$, i.e., valid initial models of the CSP-OZ-DC specification.

Thus, in order to show that actually all interpretations of *cod* satisfy *safe* one has to prove

$$\mathtt{Init} \Rightarrow \varphi_{init},$$

where `Init` is the initial constraint of *cod*, which can be done with the sequent calculus from Sect. 4.3. A second possibility is to directly show

$$\mathtt{Init} \vdash [VP]safe$$

in the calculus. Then, all interpretations in the set $[\![VP]\!](\mathcal{M}_{env} \oplus \mathcal{M}_{init})$ from inclusion (5.1) and (5.2) actually satisfy *safe*, and all interpretations of *cod* inherit this property, as desired.

Instantiating Unknown Parts

For the sake of clarity, we have have focused on the structural refinement of VA processes in this chapter and predominantly ignored assumptions on unknown parts. As motivated in the introduction in Sect. 1.2, in order to apply the VA approach, the refinement relation between processes and CSP-OZ-DC specification must particularly hold if the unknown processes in a VA process are constrained with local assumptions. Then every instantiating process of an unknown process needs to satisfy these local assumptions. Depending on the logic of the assumptions, the local assumptions are to be verified with an established verification approach for this logic. For instance, in our case studies, where the assumptions are formulated as DC properties, we apply the approach of [MFHR08] and the model checkers SLAB or ARMC to verify that the assumptions hold for the processes instantiating the unknown parts. To this end, we define the reduction of a CSP-OZ-DC specification to a sub-process:

Definition 5.5.1 (Reduction to a sub-process)
Given a CSP-OZ-DC specification c and a CSP process P, the reduction of c to P,

written $c_{|P}$, is defined as the CSP-OZ-DC specification that equals c except that the `main` *process is replaced by*

$$\texttt{main} \overset{c}{=} P,$$

and the initial schema of c is replace by [true]. The latter is necessary, because the initial constraint of the original specification does not need to be valid when considering only the sub-process P.

With the following corollary to Thm. 5.3.1 and Thm. 5.3.2, we state that the refinement relation between a VA process with local assumptions and a CSP-OZ-DC specification can be established with the refinement rules from this chapter if all instantiating processes of unknown parts fulfil the corresponding assumptions.

Corollary 5.5.2 (Refinement with assumptions over unknowns)
Let VP be a VA process with constrained unknown components $(U_1 \bullet F_1), \dots, (U_n \bullet F_n)$. Parallelism over unknowns is not allowed in VP, i.e., all unknowns occur in a sequential context. Furthermore, let cod be a CSP-OZ-DC specification such that

1. *VP with all assumptions over unknowns removed is refined by cod,*
2. *each DC counterexample trace of cod does not restrict the initial state, i.e., it begins with a true phase,*
3. *every sub-process P_i of cod that instantiates an unknown component U_i satisfies F_i. That is, $cod_{|P_i} \models F_i$, where $cod_{|P_i}$ is the reduction of cod to the process P_i.*

Then, VP is refined by cod, $VP \sqsubseteq cod$, or $VP \sqsubseteq_{inst} cod$, in the case that an instantiation relation is used.

The condition 1. can be proven with the matching rule of Sect. 5.4 or the rules of Sect. 5.3. The restriction to processes without parallelism over unknowns is important here, because parallel unknowns can introduce interferences of shared variables. These interferences cannot occur in *VP* and can possibly violate the desired safety property in instantiations of *VP*. This issue is discussed in Sect. 6.1.

For a sub-process P_i of *cod* that refines an unknown process U_i in a sequential context, every interpretation of P_i, which is known to satisfy F_i, i.e., $[\![F_i]\!] \supseteq [\![cod_{|P_i}]\!]^{\mathcal{E}}\mathcal{M}$, and which is known to satisfy the refinement condition for the unconstrained U_i, i.e., $[\![U_i]\!]\mathcal{M} \supseteq [\![cod_{|P_i}]\!]^{\mathcal{E}}\mathcal{M}$, the following holds according to the semantics of constrained unknowns

$$[\![U_i]\!]\mathcal{M} \cap [\![F_i]\!] \supseteq [\![cod_{|P_i}]\!]^{\mathcal{E}}\mathcal{M},$$

which is equivalent to

$$[\![U_i \bullet F_i]\!]\mathcal{M} \supseteq [\![cod_{|P_i}]\!]^{\mathcal{E}}\mathcal{M}.$$

Thus, $cod_{|P_i}$ actually refines the constrained unknown process $U_i \bullet F_i$ for all $i \in 1..n$ and, overall, $VP \sqsubseteq cod$.

5.6. Discussion

Using the example of CSP-OZ-DC, we have shown how a VA can be instantiated with a combined specification in such a way that safety properties proven for the VA are preserved at the level of the instantiating specification. In the following, we discuss handling of parallel CSP-OZ-DC classes, completeness of the refinement approach, and review related work.

Parallel CSP-OZ-DC Classes

The matching rule can be used to instantiate a VA process with a single CSP-OZ-DC class. Nonetheless, if a composition of several CSP-OZ-DC classes is to be analysed, there are several ways to apply the results of this chapter:

- Sometimes it is possible to reformulate the specification such that there is one class serving as a protocol class that completely implements the VA protocol. With this, the matching rule can be used to establish a refinement of the VA process.

  This procedure yields a correct refinement with respect to the refinement notion of this chapter. However, in order to check the assumptions on unknown processes, we have to impose some restrictions on the refinements of unknown processes: It must always be possible to verify assumptions on refining processes in isolation. That is, a process refining an unknown process with additional assumptions may synchronise and communicate values with further parallel components, but only if the control flow of these components cannot block behaviour that is possible when running the entire specification. This is particularly the case if the control flow of such a parallel component is restarted whenever a part is entered that refines an unknown process. By this it is impossible that the control flow blocks an execution when verifying the component in isolation but enables the execution when running the whole specification.

  For instance, this can be achieved with a parallel component that communicates only with one process refining an unknown part, no timing constraint enforces that the parallel component leaves its initial location, and the unknown part is entered only once.

  To ensure the correct behaviour in the general case, one can check a syntactic correctness criterion for each component communicating with the refining subprocess of an unknown process: if the unknown process is triggered uniquely by an event (i.e., the event only occurs when entering the process), then it is enforced by synchronisation that the control flow of all communicating parallel components are in the initial location. Note that we do not need to take care on the data here, because the assumptions need to encode all possible initial states of processes refining unknown parts.

- In general, given several parallel CSP-OZ-DC classes, one has to compute an aggregated specification process from the `main` process of all CSP-OZ-DC classes. This can be done in a straightforward way (parallel composition of all `main` processes), but one has to take care by renaming of channels that no synchronisation of private channels takes place. Moreover, between CSP-OZ-DC classes no data sharing is possible except for message parameters of synchronising events. Thus, it has to be checked (which can be done syntactically) that no variable of a different CSP-OZ-DC class is accessed in the aggregated process.

  With these preparations both approaches presented in this chapter can be applied. Either matching can be shown via Corollary 5.4.4 or the simulation relation between the VA process and the aggregated CSP-OZ-DC process can be shown in order to apply the rules 5.3.1 or 5.3.2 directly.

Completeness

The refinement rules in this chapter have been introduced to enable a simple syntactic refinement check between a VA process and a concrete CSP-OZ-DC specification. They are not designed as complete refinement rules to relate arbitrary VA processes to arbitrary instantiating processes. The reason for this is the Verification Architecture methodology in that the basic proceeding is to design a concrete model with respect to an existing architecture, or vice versa. Nevertheless, one may ask the question whether the rules presented here can also be used in a more general setting: are the rules complete in the sense that all valid refinements according to the definitions Def. 5.1.3 and Def. 5.1.4 can actually be proven with the refinement rules in Thm. 5.3.1 and Thm. 5.3.2 or the matching condition of Def. 5.4.1?

In the case of the matching rule, the answer is clearly that it is not complete because of the strict condition that the compared processes have to be nearly identical except for instantiations of the unknown parts. And the same holds for the refinement rule of Thm. 5.3.1, because condition 3. requires syntactical coincidence of the constraints of the compared processes.

But what about the generalised refinement rule? For the time being the answer is also that it is *not* complete in the form the rule is presented here. This can be seen from standard results for process or data refinement [He89, dRE98]. A counterexample is given by the following processes:

$$\begin{aligned} VP &\stackrel{c}{=} (a \to b \to P_1) \,\Box\, (a \to c \to P_2) \\ \mathtt{main} &\stackrel{c}{=} a \to \big((b \to P_1) \,\Box\, (c \to P_2)\big) \end{aligned}$$

with P_1 different from P_2. These processes are equivalent in trace semantics, but VP cannot forward-simulate `main`. The reason is that VP can indeed simulate the first a step of `main` by which

$$(b \to P_1) \sim \big((b \to P_1) \,\Box\, (c \to P_2)\big) \text{ or}$$
$$(c \to P_2) \sim \big((b \to P_1) \,\Box\, (c \to P_2)\big)$$

must be in the simulation relation $\sim$. In the case of the second line, $(b \rightarrow P_1) \Box (c \rightarrow P_2)$ can make a b step that cannot be simulated by $c \rightarrow P_2$; for the first line a c step cannot be simulated. Thus, forward-simulation is not sufficient to prove every refinement relation on processes.

However, if *backward-simulation* is considered in addition to forward-simulation the proof method becomes complete [He89, dRE98]. Backward-simulation means that every step of the concrete process can be simulated by a backward-step of the VA process. That is, a relation $\sim$ is called *backward-simulation* iff

$\overline{P} \sim \overline{Q}$ and $Q \stackrel{a}{\Longrightarrow} \overline{Q}$ implies

that there is a process P with $P \stackrel{a}{\Longrightarrow} \overline{P}$ and $P \sim Q$.

For the small example above backward-simulation can be used to show that VP (backward)-simulates `main`.

So, in order to cover more cases, proof rule Thm. 5.3.2 has to be extended such that in condition 1. the process `main` is either forward- or backward-simulated. However, even with this extension the rule is not complete, because it examines event structure and state changes separately. A counterexample is given by the following processes:

$$\begin{aligned} VP &\stackrel{c}{=} (a \rightarrow P_1) \\ \texttt{main} &\stackrel{c}{=} (a \rightarrow P_1) \Box (b \bullet \varphi \rightarrow P_2), \end{aligned}$$

where φ evaluates to false. The extended refinement rule compares the unconstrained processes via simulation on the event structure by which it can not be detected that the b event never occurs. Thus, the rule cannot prove `main` to be a refinement of VP, even though it actually is one with respect to Def. 5.1.3.

Hence, a complete proof rule for refinement needs to consider a forward- and backward-simulation defined on a transition system incorporating the state space of the VA process and the CSP-OZ-DC specification. This is not in the scope of this work, because the refinement check is aimed at a structural and not a semantical analyses of processes.

In a structural sense, the proof rule Thm. 5.3.2 extended by backward-simulation is complete, because it can be used to show the refinement relation for all VA processes and CSP-OZ-DC specifications for that the process structures fit together. This excludes refinements where the unreachability of specific branches depends on the data part and not the CSP process – such refinements can only be detected with a semantical analysis.

Related Work

We have examined a simple form of CSP-OZ-DC refinement here, because only the trace semantics of the CSP processes has been taken into account. With this, a refining process can introduce new deadlocks that do not occur in the more abstract

system. This needs to be forbidden when defining refinement for a semantics with failures or similar concepts. In [He89], the refinement of processes with failures and divergences is analysed. General results on refinement or sub-typing in CSP-OZ (also considering failures) and related formalisms (e.g., combinations of Event-B and CSP) can be found in [Fis00] and [Weh02, DW03, STW10]. The textbook of de Roever and Engelhardt [dRE98] summarises results on data refinement.

The work [Rös94] describes a stepwise-refinement approach with respect to so-called *mixed terms* of specification parts and programming notation: starting from a specification of the analysed system, specification parts are stepwise replaced, ending with a program in a CSP-like programming language such that the program is a refinement of the original specification. Besides this different focus on stepwise refinement, the approach in [Rös94] examines systems without time and does not incorporate unknown components nor combined languages like CSP-OZ-DC.

The refinement approach of the related work of D'Errico and Loreti [DL10] that we have already mentioned in Sect. 3.4.3 is also step-wise refinement based. They propose a refinement cycle in that an existing and verified, but only event-based, CCS process [Mil80] with an unknown environment can be refined with a new process. For this refining process the weakest assumption on the environment is computed such that a Hennessy-Milner property [HM85] is preserved.

6 Limitations and Extensions

I lost some time once. It's always in the last place you look for it.

(Delirium, in Season of Mists, *Neil Gaiman)*

6.1. Parallel Unknown Processes 128

6.1.1. An Automata-Based Semantics for Parallel Unknowns . . . 129

6.1.2. Translation-Based Approaches 132

6.1.3. Rely-Guarantee Reasoning for Parallel Unknowns 140

6.1.4. Instantiating Parallel Unknowns 146

6.1.5. An Interpretation-Based Semantics for Parallel Unknowns . 149

6.2. Verifying Timing Properties 158

6.2.1. Test Processes for Timing Properties 159

6.2.2. Extended Calculus for Timing Properties 161

6.3. Examining Completeness 163

6.3.1. Completeness of VA Language 164

6.3.2. Completeness of Local Assumptions 166

6.4. Complementary Decomposition Techniques 169

6.4.1. Slicing Formal Specifications 170

6.4.2. Layered Composition for Timed Protocols 171

6.1. Parallel Unknown Processes

The proof calculus as introduced in Sect. 4 contains no rules to reason directly about constrained unknown processes running in parallel with normal processes or other constrained unknown processes. Those parallel compositions are important to handle certain classes of systems, but they cause some problems. Thus, they deserve a deeper investigation, given in this section.

The main problem that occurs when composing constrained unknown processes in parallel with other processes is the interference of the temporal constraints with the operations of the other processes. Without constrained unknown processes, every state change in a CSP expression is triggered by an event and executed as single step operation or simultaneously with other events in case of synchronisation. Any possible interference of two parallel components (without unknowns) is already captured by the rule set in Chap. 4, that unfolds the parallel composition and symbolically executes every event and its corresponding operation step-by-step. In the case of a constrained unknown process, this is not possible, because the temporal constraint of an unknown process usually does not describe a single state change but a set of possible traces. Now, every execution of a parallel state change may violate the constraint. Thus, we need to handle this interference problems in our calculus to be able to treat a preferable large class of systems.

In the following, we examine two possible directions to solve this issue:

1. As a general solution, we introduce a single proof rule to handle constrained unknown processes that are translated into CSP processes without unknowns. Instead of handling the constraints of an unknown process in the calculus, they are translated into a process composed in parallel with the remaining components. Then, the standard rules from Sect. 4.3 can be used to treat the resulting CSP process.

2. We introduce proof rules that verify explicitly that the constrained unknown process does not interfere with the parallel process such that the desired property is violated. The rules are *Rely-Guarantee* based verification rules that ensure no step of one component violates important properties of the other component.

Both of these approaches are examined in the following sections. But beforehand, we need to define the semantics of constrained parallel components since in Sect. 3.1.2 parallel compositions over unknown parts were ignored. The reason is twofold: first, the semantics of an expression $\mathtt{Proc}^{(\infty)}_{\backslash A,V} \bullet F$ cannot be directly defined in terms of transition steps in a labelled transition system, because F describes entire interpretations and not single steps; second, it also cannot be (solely) defined in terms of sets of interpretations because—as we have seen in Sect. 3.2—the set of interpretations of a parallel composition cannot be computed from its constituents. This was no problem

up to here, because parallel compositions of known processes can be computed via the labelled transition systems of the processes.

To cope with the first reason, it is possible to translate the formula F in an LTS-based representation, which is essentially what we do to define the semantics. To simplify the presentation of the proof rule for parallel unknowns in the following section, we here define its semantics directly in terms of (modified) Phase Event Automata (PEA) that are used for the translation-based verification. In Sect. 6.1.5, we present a more general interpretation-based semantics for parallel unknowns that is compatible with the automata semantics.

6.1.1. An Automata-Based Semantics for Parallel Unknowns

In [Hoe06] a translation of CSP processes into PEA is given, which is also explained in Sect. 2.3. The translation uses the LTS of the CSP process and rewrites it in terms of a PEA (without timing constraints). We adapt this translation to our needs, but beforehand we examine the different notions of synchronisation in PEA and CSP.

One major difference between the semantics of PEA and CSP with data is the synchronisation on data: in contrast to the PEA semantics, data in parallel CSP processes are not synchronised. Since this only causes problems for parallel compositions, we remove parallelism—using the guarded normal form—from the process as far as possible before translating it into PEA. The only parallel compositions that remain are compositions with unknown parts.

To adequately describe the semantics with asynchronous data, a formalism between PEA (with data and fully synchronous transition behaviour) and standard timed automata (without data and synchronisation on common events) is necessary. Thus, we use PEA with a special parallel composition operator that does not synchronise on the data part. It can be applied to slightly modified PEA without stuttering transitions, which are not necessary, because the new parallel composition implicitly allows asynchronous steps of single components. The definition is similar to the standard parallel composition for timed automata and distinguishes between asynchronous steps of single components and synchronous steps of two automata that agree on an event from the synchronisation alphabet.

Definition 6.1.1 (Parallel composition without data synchronisation)
The data-asynchronous parallel composition of PEA $\mathcal{A}_l = (P_l, V_l, A_l, C_l, E_l, s_l, I_l, P_l^0)$, $l \in 1..2$*, without stutter transitions but with disjoint clock sets,* $C_1 \cap C_2 = \varnothing$*, is defined by*

$$\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2 := (P_1 \times P_2, V_1 \cup V_2, A_1 \cup A_2, C_1 \cup C_2, E, s_1 \wedge s_2, I_1 \wedge I_2, P_1^0 \times P_2^0),$$

where E is given by synchronisation and interleaving steps:

- *Synchronisation: if* $(p_l, \varphi_l, X_l, p_l') \in E_l$ *for* $l = 1, 2$*, then*

$$((p_1, p_2), \updownarrow b \wedge \varphi_1 \wedge \varphi_2, X_1 \cup X_2, (p_1', p_2')) \in E$$

for every $b \in B$.

- *Interleaving: if $(p, \varphi, X, p') \in E_1$ and $q \in P_2$, then*

$$((p, q), \bigwedge_{b \in B} \not\updownarrow b \wedge \varphi, X, (p', q)) \in E$$

and, vice versa, if $(q, \varphi, X, q') \in E_2$ and $p \in P_1$, then

$$((p, q), \bigwedge_{b \in B} \not\updownarrow b \wedge \varphi, X, (p, q')) \in E.$$

With this, we define the automata-based semantics of processes with parallelism over unknowns. We consider processes in guarded normal form having the nice property that all parallel and sequential compositions, except for compositions with unknown processes, are resolved. Then we can use the translation of [Hoe06] for CSP processes and only need to pay attention to compositions with unknown parts.

PEA semantics for parallel unknowns. Let P be a process in guarded normal form (Def. 3.3.1) with a finite process structure. We define

$$\mathcal{I} \in [\![P]\!] \quad \text{iff} \quad \mathcal{I} \text{ matches a run } \pi \text{ of } PEA_S(P), \tag{6.1}$$

where PEA_S is a PEA representing the semantics of P. It is defined by

$$PEA_S(P) := \begin{cases} PEA(P) & \text{for } P \in \{\texttt{Skip}, \texttt{Stop}, a \rightarrow Q, \\ & \qquad Q_1 \,\Box\, Q_2, Q[Rn], Q \setminus A\} \\ PEA_{\texttt{Proc}}(P) & \text{for } P = \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \\ PEA_S(\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F) \,\text{\textdegree}\, PEA_S(Q) & \text{for } P = \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \,\text{\textdegree}\, Q \\ PEA_S(\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F) \tilde{\underset{B}{\|}} PEA_S(Q) & \text{for } P = \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \underset{B}{\|} Q \end{cases}$$

The symmetric cases are omitted. See Sect. 6.1.5 for the argument that this definition coincides with the definition of Sect. 3.1.2 (for processes without parallel composition over unknowns). In this definition, unknown processes are translated into a parallel composition of two PEA using a mapping $PEA_{\texttt{Proc}}(P)$, which is defined below. Sequential composition by an unknown process with another CSP process is directly translated into a sequential composition on the PEA level. The same holds for parallel compositions with unknown components.

Processes P that do not contain unknown processes at the highest level are translated with a mapping $PEA(P)$, which basically equals the translation from CSP to PEA from [Hoe06] except that we have to integrate the automata for the unknown parts. It is based on the LTS of P, given by $LTS(P) = (Q \cup Q_{\texttt{Proc}}, L, P, \longrightarrow)$. In the case that P contains unknown processes, $Q_{\texttt{Proc}}$ with $Q_{\texttt{Proc}} \cap Q = \varnothing$ represents processes of the shape $\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \in Q_{\texttt{Proc}}$ and $\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F \boxtimes P \in Q_{\texttt{Proc}}$

for $\boxtimes \in \{\|, \Box, ;\}$ and its symmetric variants. They are treated as atomic processes that are not resolved in the LTS. Also note that the labels of the LTS are annotated events of the shape $a \bullet \varphi$. $PEA(P)$ is then given by the smallest automaton $\mathcal{A} = (Ph, SysVar \cup Const, alph(P), C, E, true, I, P)$, where

- $Ph = Q \cup \bigcup_{q \in Q_{\mathtt{Proc}}} Ph_q$, where Ph_q is the set of locations of the PEA of process q, $PEA_S(q)$. So, the set of locations of $\mathcal{A}$ comprises the set of LTS locations and the set of locations from the PEA for the unknown parts.

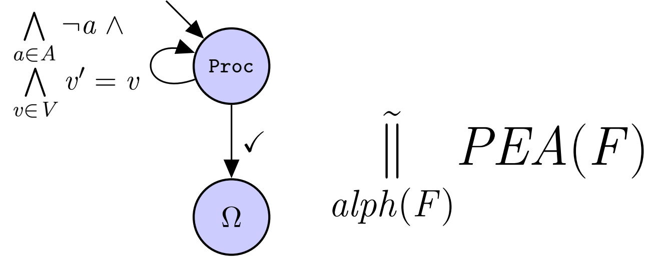
- $E = \{(q, only(a) \wedge \varphi \wedge \Xi Const, \varnothing, q') \mid q \xrightarrow{a \bullet \varphi} q', q' \in Q\} \cup \bigcup_{q \in Q_{\mathtt{Proc}}} E_q$, where E_q is the set of transitions of $PEA_S(q)$. Like in [Hoe06] $only(a)$ represents a transition where a is the only possible event.
- $C = \bigcup_{q \in Q_{\mathtt{Proc}}} C_q$, where C_q is the set of clocks of $PEA_S(q)$.
- $s = \bigcup_{q \in Q_{\mathtt{Proc}}} s_q$, where s_q is the set of state invariants of $PEA_S(q)$. All state invariants for locations in Q are set to true.
- $I = \bigcup_{q \in Q_{\mathtt{Proc}}} I_q$, where I_q is the set of clock invariants of $PEA_S(q)$. All clock invariants for locations in Q are set to true.
- Moreover, the set of system variables for the PEA is given by $SysVar$, constants $Const$ are never changing variables, the alphabet of PEA and process coincide, and the initial location is the process P itself.

For processes without unknown parts, $Q_{\mathtt{Proc}} = \varnothing$, this definition equals the definition of Hoenicke in [Hoe06] except that constrained events $a \bullet \varphi$ occur in the LTS. For processes with unknown parts, the definition glues the automata for the unknown parts, $PEA_{\mathtt{Proc}}(q)$ for $q \in Q_{\mathtt{Proc}}$, into the PEA of the process part without unknowns, which is constructed according to the standard translation for CSP. This definition is well-defined, because we restrict ourselves to processes with a finite structure here (for instance, processes of the shape $P \stackrel{c}{=} \mathtt{Proc}_{\setminus A, V} \bullet F \underset{B}{\|} P$ are not allowed).

Translation of unknown parts into PEA. Hence, we now define the translation of unknown processes into PEA in such a way that the parallel components are correctly synchronised.

Figure 6.1 defines the translation from unknown processes into PEA. The constraining formula F is translated into an equivalent representation in terms of PEA—such a translation exists by requirement. In addition, we demand that all transitions of $PEA(F)$ are event triggered. The automaton of the formula $PEA(F)$ and a second automaton, representing the PEA for $\mathtt{Proc}_{\setminus A, V}$, are synchronised over the entire alphabet of F, i.e., the alphabet of its automaton $alph(F) := alph(PEA(F))$. This second automaton consists of two locations: one representing the unknown process and one representing the terminated process Ω. The former location has a transition

$PEA_{\texttt{Proc}}(P_1) :=$



$PEA_{\texttt{Proc}}(P_2) :=$

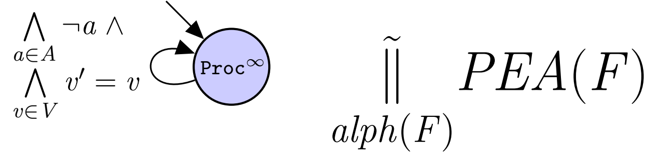


Figure 6.1.: Translation from $P_1 = \texttt{Proc}_{\setminus A,V} \bullet F$ and $P_2 = \texttt{Proc}^\infty_{\setminus A,V} \bullet F$ into PEA

representing a data change executed by the process itself: no occurrence of events from A and no change of variables in V are allowed. Thus, the parallel composition may perform arbitrary events that are not in the set A as long as variables from V are not changed. If the event is in the alphabet of $PEA(F)$, it can only occur if allowed by $PEA(F)$. If an event is not restricted by $PEA(F)$, the automaton may perform arbitrary non-synchronised steps over this event.

The automaton for $\texttt{Proc}^\infty_{\setminus A,V}$ equals that of $\texttt{Proc}_{\setminus A,V}$ but without the Ω location, representing the terminated process.

6.1.2. Translation-Based Approaches

A possible solution to solve properties over parallel unknown components with additional constraints is to apply an automata-theoretic approach. That is, instead of solving real-time constraints directly in the calculus, we translate the constraints together with all parallel components into a single CSP process with data—without timing properties and unknown components. This has the drawback that the translation is not compositional, because the entire parallel composition is unfolded, including all timing properties in the range of the current process.

Example 6.1.2. An example demonstrating the problems of parallel processes constrained by timed DC properties is

$$(\texttt{Proc} \bullet \neg(\ell \leq 5 \frown \updownarrow a \frown true)) \underset{a,b}{\|} a \rightarrow b \rightarrow Bad \underset{a,b}{\|} (\texttt{Proc} \bullet \neg(\boxminus b \wedge \ell > 2 \frown true)). \tag{6.2}$$

The process *Bad* represents an arbitrary process that causes undesired behaviour violating a safety property. The left DC formula demands that for 5 time units no a event occurs while the right DC formula requires the occurrence of a b event within 2 time units. Due to these DC formulae, b has to occur before event a, which contradicts the middle CSP process. Hence, the *Bad* process cannot be reached, and the system is safe. This property depends on the concrete time constants used in the DC formulae. If the time constant in the right formula is replaced by a value larger than 5, then the process *Bad* becomes reachable.

The problem with suchlike processes is the interdependence of all of the three sub-processes that has to be solved by any approach to compositionally reason on processes. Of course we would prefer to follow the deductive approach from Chap. 4, but then we would have to find a way to step-wise reduce the expression (6.2) to a simpler expression such that its meaning its preserved. Candidates for such a reduction in (6.2) are the DC formulae and the prefix process.

The problem becomes even more difficult if we slightly extend (6.2) to

$$\Big((\mathtt{Proc} \bullet \neg(\ell \leq 5 ^\frown \updownarrow a ^\frown \mathit{true}) \wedge \boxminus b) \underset{a}{\|} a \rightarrow b \rightarrow \mathit{Bad}\Big) \tag{6.3}$$

$$\underset{b}{\|} \Big(\mathtt{Proc} \bullet \neg(\boxminus b \wedge \ell > 2 ^\frown \mathit{true}) \wedge \boxminus a\Big), \tag{6.4}$$

because now it is not possible to compute the interaction of both of the DC formulae before taking the prefix process expression into account. Instead one has to first compute the interaction of the prefix expression with the left process. That is, if we want to reduce these processes in a semantic-preserving way, we need a representation for the process structure and the timing information. Neither can be dropped here, because both are necessary for synchronisation with the right `Proc` expression.

So, any possible reduction of such a process expression must lead to a simpler representation of (6.3) while preserving all relevant information for the synchronisation with the process in (6.4), which is nearly all information contained in the sub-processes. Such a reduction could for instance be a timed dependency reduction stating that event b always occurs before event a. In the example, this works only in the case of (6.2) (when computing the parallel composition of both unknown processes) and not for the parallel composition in (6.3) and (6.4), where we need the time constraints for both synchronisations. This leads to the conclusion that suchlike process expressions, with a large degree of interdependence, cannot effectively be simplified in a compositional way. However, we could transform the entire process expression into a representation that is easier to handle. For instance, this could be a unified representation not combining different formalisms, e.g., in this case, a reduction to a CSP process or a DC formula.

Overall, this result is not very surprising, because it is always the case in decompositional verification that a system, where a property crucially depends on every single action (and its timed behaviour) of a parallel composition, cannot be further

decomposed. It follows that every approach to *compositionally* solve these parallel compositions like in Sect. 6.1.3 can only provide a solution for dedicated situations. Thus, as a solution for the general case, we provide a proof rule that transforms parallel process expressions with unknown parts into a single process expression without parallelism and without timing constraints. This process can then further be treated with the calculus rules from Sect. 4.2. A second solution, which we do not examine here, is the translation of the entire process expression into a formula that can be solved with a verification technique for the corresponding logic.

Proof Rule Based on Translation

With the following generic proof rule, general CSP processes with constrained unknown parts can be resolved into a CSP process without further timing constraints. The premise of the rule can then be further processed with the proof rules from Sect. 4.2.

$$\frac{[\mathit{ProcFree}(P)]\gamma}{[P]\gamma} \qquad \text{(parallel uproc)}$$

The process expression $\mathit{ProcFree}(P)$ is a process with data but without parallel compositions, and it is a satisfiability equivalent representation of P in the sense that for all formulae γ and models $\mathcal{M}$

$$\exists \mathcal{I} \in [\![P]\!]\mathcal{M} \bullet \mathcal{I} \models \gamma \quad \text{iff} \quad \exists \mathcal{I} \in [\![\mathit{ProcFree}(P)]\!]\mathcal{M} \bullet \mathcal{I} \models \gamma. \tag{6.5}$$

The rule is similar to rule (process equivalence) with the difference that the processes of conclusion and premise are not exactly equivalent but satisfiability equivalent. It is necessary to take account of satisfiability equivalence in order to cover the case that the interpretations of P are restricted by timing constraints whereas $\mathit{ProcFree}(P)$ has the same untimed interpretations as P without further timing restrictions.

Theorem 6.1.3 (Soundness of rule (parallel uproc)**)**
Rule (parallel uproc) *with side-condition* (6.5) *is sound.*

Proof. We show local soundness, i.e., that $\mathcal{M} \models [\mathit{ProcFree}(P)]\gamma$ implies $\mathcal{M} \models [P]\gamma$ and vice versa. For the first implication, this is the case, because $\mathcal{M} \models [\mathit{ProcFree}(P)]\gamma$ means that

$$\text{for all } \mathcal{I} \in [\![\mathit{ProcFree}(P)]\!]\mathcal{M} \text{ holds } \mathcal{I} \models \gamma. \tag{6.6}$$

If we now assume that $\mathcal{M} \not\models [P]\gamma$, then there is an interpretation $\mathcal{J} \in [\![P]\!]\mathcal{M}$ with $\mathcal{J} \models \neg\gamma$. But then we can apply equivalence (6.5) and get that there also is an interpretation $\overline{\mathcal{I}} \in [\![\mathit{ProcFree}(P)]\!]\mathcal{M}$ with $\overline{\mathcal{I}} \models \neg\gamma$, which contradicts (6.6). The other direction of the equivalence is symmetric. □

Construction of $ProcFree(P)$

Even though the proof rule introduced above can be applied to arbitrary processes satisfying condition (6.5), we now show how to construct a satisfiability equivalent process without unknown processes for every process P. The construction is generic, because it can be applied to all unknown processes that are constrained by a temporal logic that can be translated into timed automata [AD94]. As timed automata representation we use PEA.

The construction shows that there actually is a process $ProcFree(P)$ for every P as long as the constraining logic for unknown processes in P can be translated into PEA, like it is the case for DC formulae [Hoe06]. If P is already a process without unknown processes, then $ProcFree(P)$ is the process where every parallel composition is equivalently replaced by a choice of processes like for rule (parallel$_\Box$) on page 75.

The construction of $ProcFree(P)$ can be sketched as follows.

1. The CSP process with constrained unknown parts is translated into its PEA-based semantics according to the definitions above.

2. To this automaton the region construction of [AD94] is applied and by this all clock constraints are removed.

3. The resulting finite state automaton, where all events are annotated by constraints, is translated back to a CSP process with data.

These steps are examined in detail:

PEA representation. The process P is carried over to its semantics in terms of PEA with the mapping $PEA_S(P)$ (Sect. 6.1.1). It holds by definition that

$$\mathcal{I} \in [\![P]\!] \text{ iff there is a run } \pi \text{ of } PEA_S(P) \text{ matching } \mathcal{I}. \tag{6.7}$$

Region automaton for PEA with infinite data. In the next step, the clock constraints in PEA_S, which have been introduced during the translation of the constraints of unknown processes, need to be removed. Hence, we apply the region construction of [AD94] to translate the infinite-state PEA into a finite LTS without clocks. The original region construction for timed automata is without data, but can be extended to automata with finite data as done similarly for the model checker Uppaal [LPY97, Möl02]. So, the main idea to cope with the possibly infinite data constraints of CSP and PEA is to perform the standard region construction while treating the data constraints as event annotations. Afterwards, the resulting automaton without timing constraints is translated back to CSP, and we handle the data constraints on this level. This is possible, because the timing constraints of PEA usually do not depend on data constraints with the exception of timing parameters. Thus, we only need to dispense with clock constraints over timing parameters (cf. Def. 2.3.1) that actually compare clocks with data.

Another syntactical restriction we impose is that we only consider PEA without state invariants (clock invariants are of course allowed), which slightly simplifies the construction and which enables the back translation into CSP with data. This is not a significant restriction, because every PEA (after computing the parallel product) can be translated into a PEA without state invariants by shifting the invariant constraints to incoming transitions of the corresponding location; for initial locations with state invariants a new dummy initial location is used to set the invariant constraint when entering the original initial location.

To sum up, we examine PEA without state invariants where every state change is executed at transitions triggered by events. We can see the constraints as syntactic event annotations. To give an example, the PEA transition

$$(p, \updownarrow a \wedge \updownarrow b \wedge x' = x + 1 \wedge c < 15, X, q),$$

where x and x' are integer variables, a and b are event variables, c is a clock, and X a set of clocks, can be rewritten to a transition

$$\big(p, \updownarrow (\updownarrow a \wedge \updownarrow b \wedge x' = x + 1) \wedge c < 15, X, q\big).$$

In this transition, $\updownarrow(\updownarrow a \wedge \updownarrow b \wedge x' = x + 1)$ is an *annotated event*, where $\updownarrow a \wedge \updownarrow b \wedge x' = x + 1$ is regarded as the event name, and the transition can be considered as standard timed automata transition. For simplicity, we omit the duplicate event arrows and write $\updownarrow a \wedge \varphi$ instead of $\updownarrow(\updownarrow a \wedge \varphi)$. The same event $\updownarrow a$ with a different annotation $\varphi \neq \psi$ is considered as a different event, i.e., $\updownarrow a \wedge \varphi \neq \updownarrow a \wedge \psi$. Note that we do not need events for synchronisations anymore, because we have already computed the parallel composition of all parallel components. We formally define:

Definition 6.1.4 (TA corresponding to PEA)
For a PEA $\mathcal{A} = (P, V, A, C, E, s, I, P^0)$ the timed automata representation of $\mathcal{A}$, $TA(\mathcal{A})$, is a PEA without data defined by

$$TA(\mathcal{A}) := (P, \varnothing, \overline{A}, C, \overline{E}, \varnothing, I, P^0).$$

For simplicity we assume without loss of generality that all guards at transitions are conjunctions, which can always be achieved by transforming the guard into disjunctive normal form and removing every disjunction by splitting the transition. Particularly, every clock constraint is then a simple conjunction of (possibly negated) comparisons according to the definitions in Sect. 2.3.

With this we define

$$(p, (\updownarrow \psi) \wedge \theta, X, q) \in \overline{E} \text{ and } \psi \in \overline{A} \quad \text{iff}$$

$$(p, \varphi, X, q) \in E,$$

where φ can be split into one part with events and data constraints and one clock constraint, i.e., $\varphi = \psi \wedge \theta$. We additionally demand $A \cap \overline{A} = \varnothing$.

Lemma 6.1.5 (Runs of PEA and TA)
There is an interpretation $\mathcal{I}$, with untime($\mathcal{I}$) = $\langle \mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_n \rangle$ that matches a run $\pi = \langle (p_0, \psi_0, \mathcal{M}_0, \eta_0, t_0), \dots, (p_n, \psi_n, \mathcal{M}_n, \eta_n, t_n) \rangle$[1] *of TA($\mathcal{A}$) and that additionally respects the state changes according to the ψ_i, i.e., $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \psi_i$ for $i \in 0..n$, iff there is a corresponding run $\overline{\pi} = \langle (p_0, E_0, \overline{\mathcal{M}}_0, \eta_0, t_0), \dots, (p_n, E_n, \overline{\mathcal{M}}_n, \eta_n, t_n) \rangle$ of $\mathcal{A}$ that matches $\mathcal{I}$, with $\mathcal{M}_i(x) = \overline{\mathcal{M}}_i(x)$ for all $x \notin \overline{A}$.*

Proof. For the "only if" case, we only have to show that $\overline{\pi}$ actually is a run of $\mathcal{A}$, which can be shown by proving all conditions of Def. 2.3.4: the events in the E_i, which correspond to the events of the constraint ψ_i, can occur due to the assumption $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \psi_i$. Since π is a run of $TA(\mathcal{A})$, we know that there is a transition $(p_i, \updownarrow(\psi_i) \wedge \theta_i, X, q_i)$ for every i, which implies $\mathcal{M}_i \cup \mathcal{M}'_{i+1}, \eta_i + t_i \models \theta_i$. Together with the assumption $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \psi_i$, we can conclude $\mathcal{M}_i \cup \mathcal{M}'_{i+1}, \eta_i + t_i \models \psi_i \wedge \theta$, because the ψ_i do not contain clock constraints by construction. The remaining conditions of Def. 2.3.4 are true since $\mathcal{A}$ and $TA(\mathcal{A})$ coincide on all clock constraints and invariants.

The "if" case follows, because $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \psi_i$ due to the semantics of PEA. Due to $\mathcal{M}_i \cup \mathcal{M}'_{i+1}, \eta_i + t_i \models \psi_i \wedge \theta$, the clock constraints are not violated by which we get the corresponding run π of $TA(\mathcal{A})$. The condition $\mathcal{M}_i(x) = \overline{\mathcal{M}}_i(x)$ for all $x \notin \overline{A}$ ensures that the $\overline{\mathcal{M}}_i$ from the run $\overline{\pi}$ do not incidentally prohibit the occurrences of the compound events from $\overline{A}$, which do not occur in $\mathcal{A}$. ☐

In [AD94] it is shown that location reachability in timed automata is decidable. This is achieved by translating the timed automaton, having an infinite state space due to real-valued clocks, into an equivalent (with respect to location reachability) representation without clocks. The key observation of this construction is that it is not important to know the exact value of a clock as long as it is known in which time interval a clock value can be and how it relates to other clocks (lesser, equal, or greater). To this end, the time line is covered by a finite number of intervals (or *regions*) up to the greatest time constant occurring in the automaton. By this, an equivalence relation on the clock values is constructed. This equivalence relation can be used to define for a given timed automata $\mathcal{A}$ the so-called *region automaton for* $\mathcal{A}$, denoted $R(\mathcal{A})$, having a finite state space. For simplicity, we consider the region automaton to be a finite automaton accepting runs over configurations

$$(p, Y, [\eta]) \in P \times \mathbb{P}A \times \mathit{Region},$$

where $[\eta] \in \mathit{Region}$ denotes an equivalence class on clock valuations.

We omit the standard construction here and refer to [AD94] and [OD08] for an exact description. We only cite the classical result for standard timed automata that accordingly also holds for the timed automata representation of PEA.

[1] Note that the $\mathcal{M}_i$ of the run can be chosen arbitrarily for all data except for events, because $TA(\mathcal{A})$ does not contain any constraint over data variables.

Theorem 6.1.6 (Reachability in region automaton)
Given a timed automaton $\mathcal{A}$, that is a PEA without data variables and state invariants as in Def. 6.1.4, then there is for every run

$$\langle (p_0, E_0, [\eta_0]), \ldots, (p_n, E_n, [\eta_n]) \rangle$$

of $R(\mathcal{A})$ a corresponding run

$$\langle (p_0, E_0, \mathcal{M}_0, \eta_0, t_0), \ldots, (p_n, E_n, \mathcal{M}_n, \eta_n, t_n) \rangle$$

of $\mathcal{A}$ and vice versa.

So, using this theorem we can decide whether a specific run of the system is prohibited by its clock constraints or not. It is not revealed whether the run is allowed by the data constraints of the system. For this reason, the region automaton is translated back to a dCSP expression such that we can use the approach of Sect. 4.2 to reason about the data part.

Translation to CSP with data. A translation from state machines into CSP can be found in, e.g., [RW03]. Here the translation is simpler, because the region automaton is just a finite automaton without parallel or sequential components and without compound states.

The translation is given by a mapping *peq* assigning CSP process equations to locations of the automaton. Let $succ(l) \in \overline{A} \times P$ be the set of successors of location l, consisting of a pair of the event and the successor location for every outgoing transition of l. Note that for the region automaton the set of locations P consists of tuples $(p, [\eta])$ over a symbolic location p and an equivalence class of clocks. The set $succ_\tau(l)$ contains all successors that can be reached via an empty transition, that is not triggered by an event. Then a translation into a process expression is given by the process equation system *peq*: for a location l without outgoing transitions, i.e., $succ(l) \cup succ_\tau(l) = \varnothing$, $peq(l)$ is defined by

$$peq(l) := (P_l \overset{c}{=} \texttt{Stop})$$

and for all other locations by

$$peq(l) := \Big(P_l \overset{c}{=} (\ \Box_{(\updownarrow a \wedge \psi, m) \in succ(l)}\ a \bullet \psi \to P_m) \sqcap (\ \textstyle\sqcap_{m \in succ_\tau(l)} P_m)\Big)\,. \qquad (6.8)$$

Since we use trace semantics of CSP processes in this work, $\sqcap$ and $\Box$ cannot be distinguished. We consider `Stop` to be the neutral element of both choice operators. Thus, if $succ(l)$ or $succ_\tau(l)$ are empty, the corresponding branch in (6.8) disappears, because

$$P \,\Box\, \texttt{Stop} = P \quad \text{and} \quad P \sqcap \texttt{Stop} = P$$

in trace semantics (the right equation is wrong in failures semantics). Moreover, we have slightly simplified the presentation here by assuming that only a single event a

occurs for the transitions in (6.8). In the general case, we need to consider compound events representing complex event expressions over more than one event, similarly to Def. 6.1.4. This simplification is permissible since the events are not used anymore for synchronisations.

The translation of a region automaton R into a process $CSP(R)$ is then defined by

$$CSP(R) := \sqcap_{l \in P^0} P_l,$$

where P^0 is the set of initial locations of R.

Lemma 6.1.7 (Runs of CSP process and region automaton)
Let $CSP(R)$ be a CSP process according to the construction above such that every event a_i is constrained by a formula ψ_i. Then, there is an interpretation $\mathcal{I} = \langle \mathcal{M}_0, a_0, \mathcal{M}_1, a_1, \dots \rangle$ of $CSP(R)$ iff there is a corresponding run

$$\langle (p_0, (\updownarrow a_0 \wedge \psi_0), [\eta_0]), (p_1, (\updownarrow a_1 \wedge \psi_1), [\eta_1]), \dots \rangle$$

of R and $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \updownarrow a_i \wedge \psi_i$ for all i.

Proof. The existence of a corresponding run of R follows from the construction of $CSP(R)$. The second condition $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \updownarrow a_i \wedge \psi_i$ is a direct conclusion from the semantics of CSP with data. Vice versa, the existence of an interpretation $\mathcal{I}$ follows from the CSP semantics in Sect. 3.1.1. □

The following corollary shows that these several translation steps can be used to translate a process with arbitrary unknown parts constrained by a real-time logic into an untimed representation that is equivalent to the original process (with respect to a desired untimed property). Hence, a so-defined process *ProcFree* can be used to apply rule (parallel uproc) in order to reduce a process with constrained unknown parts.

Corollary 6.1.8 (Construction of $ProcFree(P)$)
If we set $ProcFree(P) := CSP(R(TA(PEA_S(P))))$, then condition (6.5) *is valid.*

Proof. We apply the Lemma 6.1.7, Thm. 6.1.6, and Lemma 6.1.5 to get the desired result. Consider $\mathcal{I} \in [\![ProcFree(P)]\!]\mathcal{M}$ with $untime(\mathcal{I}) = \langle \mathcal{M}_0, \dots, \mathcal{M}_n \rangle$.

1. We apply Lemma 6.1.7 and get a run π of $R(TA(PEA_S(P)))$ with side-condition $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \updownarrow a_i \wedge \psi_i$.

2. Applying Thm. 6.1.6 to π, we get a corresponding run $\overline{\pi}$ of $TA(PEA_S(P))$.

3. With the side-condition $\mathcal{M}_i \cup \mathcal{M}'_{i+1} \models \updownarrow a_i \wedge \psi_i$, Lemma 6.1.5 yields an interpretation $\overline{\mathcal{I}}$ matching a run of $PEA_S(P)$ with $untime(\overline{\mathcal{I}}) = \langle \overline{\mathcal{M}}_0, \dots, \overline{\mathcal{M}}_n \rangle$ and $\mathcal{M}_i(x) = \overline{\mathcal{M}}_i(x)$ for all $x \notin \overline{A}$.

4. Due to (6.7), we can conclude $\overline{\mathcal{I}} \in [\![P]\!]$.

Since the untimed sequences of $\mathcal{I}$ and $\overline{\mathcal{I}}$ are the same except for events in $\overline{A}$, which do particularly not occur in formula γ from condition (6.5), both interpretations cannot be distinguished by untimed formulae γ. Since all of the used lemmas and theorems describe equivalences, they can also be applied in the other direction to derive the equivalence in (6.5). □

The translation steps to construct *ProcFree*(P) consist mainly in simple rewritings of the process to bring it into a structure allowing for applying the region construction. The costly parts are the unfolding of parallel compositions in the first step, when translating the process into a PEA, and the region construction, which is exponential in the number of clocks [AD94, OD08].

Solution without region construction. [HM05a] and [Hoe06] proposed a verification approach for PEA that is not based on the region construction but on a translation of PEA into transition constraint systems (TCS) that can be verified using the model checkers ARMC [PR07] or SLAB [BDFW07]. This translation can also be used as an alternative to the region construction when generating an untimed process *ProcFree*(P) for a process with unknowns P. The basic idea is the same as before: the constrained unknown parts as well as the remaining processes are translated into PEA, and the parallel composition has to be computed, i.e., $PEA_S(P)$. But instead of the region construction, the translation into TCS is applied, by which the implicit notion of a clock is replaced by arithmetical operations on real-valued variables. This means that the real-valued variables representing the time are not automatically increased, but instead are simultaneously increased by an explicit time-progress step. For details of this translation see [Hoe06]. Afterwards, the TCS needs to be translated back to a CSP expression with data that can be verified using the calculus of Sect. 4.2, which can be done similarly to the translation of the region automaton into CSP, $CSP(R)$.

The advantage of this procedure is that it avoids the exponential structural blow-up of the process equation system, which is a result of the region construction. However, it introduces new real-valued variables, representing the time progress, and arithmetical operations over them, which have to be handled within the sequent calculus proof. Thus, for generating *ProcFree*(P) we have the choice between the region variant resulting in a complex process structure with simpler data constraints as opposed to the TCS variant with a simpler process structure but more complex data constraints, where the time progress has to be explicitly computed in a sequent proof. We have not yet compared which variant yields better results under which circumstances.

6.1.3. Rely-Guarantee Reasoning for Parallel Unknowns

After having introduced a general but not very efficient approach for solving constrained parallel unknowns, we examine situations, where parallel unknowns can directly be handled using proof rules.

As already mentioned earlier, solving properties over parallel compositions with unknown parts is difficult, because of the interference of the components, which can always occur in systems with shared variables. That is, a parallel unknown process may execute an arbitrary state change over a shared variable at any time, and by this, it may influence parallel components such that a safety property is violated. Thus, proof rules handling properties over parallel unknowns need to cope with these mutual change of system variables. A classical solution from the area of program verification of parallel programs is the use of interference freedom rules like in the approach of Owicki and Gries [OG76a, OG76b, AdBO09]. With those rules, it is explicitly proven that every state change of one component never influences the required properties of all other components. One major drawback of this approach is that it only works for closed systems, in which every component is known in advance, because one needs to check that every executed state change does not falsify relevant properties at any point of the execution. One approach to overcome this problem is to apply the Assume-Guarantee paradigm for shared variables (also called *Rely-Guarantee* method) [Jon81, MC81, Jon83, dRdBH+01]. Implicitly, this approach also implies a kind of interference freedom of the parallel components, which is not formulated by a global property over all components but as a property of every single component. We adapt this approach in the context of CSP processes with unknown parts.

To begin with, we look at straightforward rules for parallel unknown processes and argue why these attempts cannot work. When examining a property over a parallel composition like

$$\Delta \vdash [\mathtt{Proc}_{\backslash A_1, V_1} \bullet F_1 \underset{B}{\parallel} \mathtt{Proc}_{\backslash A_2, V_2} \bullet F_2]\gamma, \tag{6.9}$$

one usually would like to have a proof rule that can be treated completely within the sequent calculus of Sect. 4.3. That means that the rule needs to decompose the property or the process into smaller parts that can be handled in the calculus. This could be for instance a rule like

$$\frac{\Delta_0 \vdash \Gamma_0 \quad \Delta_1 \vdash [\mathtt{Proc}_{\backslash A_1, V_1} \bullet F_1]\gamma_1 \quad \Delta_2 \vdash [\mathtt{Proc}_{\backslash A_2, V_2} \bullet F_2]\gamma_2 \quad \gamma_1 \vee \gamma_2 \Rightarrow \gamma}{\Delta \vdash [\mathtt{Proc}_{\backslash A_1, V_1} \bullet F_1 \parallel \mathtt{Proc}_{\backslash A_2, V_2} \bullet F_2]\gamma, \Gamma}.$$

This inaccurate rule combines local properties γ_1 and γ_2 of the unknown processes to conclude the desired global γ. This cannot work in the general case, because the two unknown processes may access shared variables. Moreover, as the logic of F_1 and F_2 is not known in advance, we do not know what is expressible by the assumptions. We consider an example to clarify this:

Example 6.1.9. The dCSP formula

$$p \vdash [\mathtt{Proc}_{\backslash \varnothing, \varnothing} \bullet \Box p ||| \mathtt{Proc}_{\backslash \varnothing, \varnothing} \bullet \Box p]\Box p \tag{6.10}$$

expresses that when starting with a valuation s.t. p holds and executing two parallel unknown processes ensuring that p holds, the desired property is that p always holds.

Unfortunately, even this property does not hold for all possible instantiations of the unknown parts. A counterexample is, e.g., the process

$$Q \overset{c}{=} (a_1 \bullet p \wedge b' \wedge p') \to \big(((a_2 \bullet b \wedge p' \wedge \neg b') \to \texttt{Skip}) \,\Box\, ((a_3 \bullet \neg b \wedge \neg p') \to \texttt{Skip})\big)$$

as instantiation process for both unknown processes. When executing such a process Q in isolation, then p is always true. But as soon two instances of this process run in parallel, there is an execution such that the first process sets b to false, which enables to second branch of the choice for the other process. So, the second process may set p to false.

Hence, the example shows that the instances of the unknown processes in (6.10) interfere in an unpredictable way, because the problem-causing property b does not occur in (6.10). For this reason, to prove properties over such parallel compositions, we need to examine every single transition of the components (instead of traces like in the given naïve rule) to exclude property-violating interference at each possible data change. In the framework of Owicki and Gries [OG76a], this is done by explicitly proving non-interference for each transition step[2]. But this approach is not possible for unknown processes, which are not known in advance. Instead we follow the Rely-Guarantee (R-G) approach [dRdBH+01] and impose a Rely-Guarantee condition for every transition of the unknown process, demanding that when every transition up to a given point in time satisfies a *rely* condition, then the next transition satisfies a *guarantee* condition. Thus, we require this property to be checked in the constraining logic of the unknown processes, i.e., a property like (6.9) can be proven to actually hold for all instantiations of unknown processes if F_1 and F_2 are formulated as appropriate Rely-Guarantee formulae. Whether this is possible depends on the constraining logic for unknowns, and we will see below how R-G formulae can be formulated in our standard logic DC or in terms of CSP processes without unknowns.

The next section presents four proof rules to verify that parallel components do not interfere in such a way that a desired property is violated.

Proof Rules

To begin with, we have a few simple rules that cover trivial situations—similarly to rule (assumption axiom)—where the system is blocked due to necessary synchronisations. In this case, every state formula δ is valid. So, with $a \in A \cap B$ the following two rules reflect this fact, for state formulae and for the temporal case.

$$\frac{}{[a \to Q \parallel_B \texttt{Proc}^{(\infty)}_{\setminus A, V} \bullet F]\delta} \qquad \text{(parallel uproc axiom)}$$

$$\frac{\varphi}{[a \to Q \parallel_B \texttt{Proc}^{(\infty)}_{\setminus A, V} \bullet F]\Box\varphi} \qquad \text{(parallel uproc}_\Box\text{)}$$

[2]In Example 6.1.9 such an non-interference check fails, because the second branch of the choice in Q actually does interfere with the desired property.

These rules over prefix processes can be easily expanded to the general guarded case. Instead of a single prefix process, a choice of guarded processes is examined. For instance, instead of checking only the a event in the left premise of the rules one needs to check all possible events that can occur in the left process. As parallel composition is commutative, all of these rules (and likewise the following rules) also hold for the symmetric case.

The next rule handles the case that an unknown process is actually independent from the second parallel component, i.e., a desired property can be proven without checking the constrained unknown part. So with a concrete process P without unknown parts and $SysVar(P) \cup SysVar(\gamma) \subseteq V$—that is $\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F$ does not change any symbol in P and γ—the rule is as follows.

$$\frac{\Gamma \vdash [P]\gamma, \Delta}{\Gamma \vdash [P \underset{B}{\parallel} \texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F]\gamma, \Delta} \qquad \text{(independent uproc)}$$

Note that parallel compositions satisfying the side-condition are not necessarily independent from each other, because the unknown process could potentially forbid some runs of P leading to bad behaviour. Therefore, the rule is not locally sound.

After these rules for handling special situations, we now introduce a general R-G rule to solve properties over parallel processes, that may also contain unknown parts.

$$\frac{\Gamma \vdash \psi \wedge \sigma, \Delta \quad \psi \vee g_2 \vdash r_1 \quad \psi \vee g_1 \vdash r_2 \quad g_1 \vee g_2 \vdash \overline{\gamma}}{\Gamma \vdash [P_1 \parallel P_2]\gamma, \Delta} \qquad \text{(R-G uproc)}$$

Here, γ is either defined by $\gamma = \delta$, i.e., it is a state formula, or by $\gamma = \Box\sigma$, i.e., it is a temporal property, and $\overline{\gamma}$ is given by $\overline{\gamma} = \delta$ or $\overline{\gamma} = \sigma$, respectively[3]. The side-condition of this rule is that P_1 and P_2 satisfy the Rely-Guarantee criterion: if a state satisfies a rely condition r_i, then after the next state change of P_i the guarantee condition g_i is satisfied. Furthermore, g_i is stable, i.e., if g_i holds, then g_i is still valid after every state change of P_i. Formally: for every interpretation $\mathcal{I}_i \in [\![P_i]\!]$ with $untime(\mathcal{I}_i) = \langle \mathcal{M}_0, \ldots, \mathcal{M}_n \rangle$ it holds for $i \in 1..2$ and $j \in 1..n$

$$\mathcal{M}_{j-1} \models r_i \vee g_i \quad \Rightarrow \quad \mathcal{M}_j \models g_i. \tag{6.11}$$

Note that we do not explicitly distinguish between transitions of the process itself and its environment process. If $\gamma = \Box\sigma$, then we need to show in the left premise that σ is valid at the beginning of the execution of $P_1 \parallel P_2$. The current context formulae Δ, Γ are also considered when proving the initial condition ψ (but not in the remaining premises).

The rule is motivated by the parallel composition rule for Rely-Guarantee reasoning [dRdBH$^+$01], but it is also similar to the Rely-Guarantee rule of [Sti88] in that the rely and the guarantee conditions are formulated as state predicates and not as transition predicates over primed and unprimed variables. We could have defined the

[3] In the case $\gamma = \delta$, the σ occurring in the rule is not relevant, i.e., $\sigma = \text{true}$.

proof rule in terms of transition predicates likewise, but with the state predicate approach, definitions and proofs become slightly simpler, and state predicates are more convenient when defining the rely and guarantee conditions in terms of DC formulae.

In the premise of the rule, one needs to show that the guarantee condition g_1 of one component implies the rely condition r_2 of the other and vice versa. In combination with the initial condition, which holds in the current context Δ, Γ and which also implies the rely conditions, this implies that one of the guarantees is always true. Finally, one need to prove that each of the guarantee conditions implies the desired property.

Theorem 6.1.10 (Soundness of proof rules for unknown processes)
The presented proof rules (parallel uproc axiom) *up to* (R-G uproc) *with the corresponding side-conditions are sound.*

Proof.

Rule (parallel uproc axiom): The rule is correct, because there is only one interpretation $\mathcal{I} \in [\![a \to Q \underset{B}{\|} \texttt{Proc}^{(\infty)}_{\backslash A, V} \bullet F]\!]\mathcal{M}$ for all models $\mathcal{M}$, and this interpretation is of the shape $\mathcal{I} = \langle \mathcal{M} \rangle$ since no synchronisation is possible. $\mathcal{I}$ has no terminating model, so $\mathcal{I} \models \delta$.

Rule (parallel uproc$_\Box$): With the same argument as in the previous case we get that the only possible interpretation of the parallel composition is $\mathcal{I} = \langle \mathcal{M} \rangle$. Hence, $\mathcal{M} \models \varphi$ iff $\mathcal{I} \models \Box\varphi$ which proves the rule sound.

Rule (independent uproc): We give the proof sketch here: For an interpretation

$$\mathcal{I} \in [\![P \underset{B}{\|} \texttt{Proc}^{(\infty)}_{\backslash A, V} \bullet F]\!]\mathcal{M} \text{ with } \mathcal{I} = \langle \mathcal{M}_0, a_0, \dots \rangle,$$

the projection $\mathcal{I}_{|V}$ to symbols in V is considered, i.e., $\mathcal{I}_{|V} = \langle \mathcal{M}_{0|V}, a_0, \dots \rangle$, where $\mathcal{M}_{i|V}(v) = \mathcal{M}_i$ for $v \in V$, and $\mathcal{M}_{i|V}(x)$ is undefined for all other symbols $x \notin V$. Then, for all single transition steps a_i of the unknown process $\mathcal{M}_{i|V} = \mathcal{M}_{i+1|V}$, because the unknown process does not change variables in V. These stutter steps are removed from the interpretations which results in an interpretation of P. To that the premise is applied by which property γ is satisfied for the modified interpretation. Since γ contains only symbols from V, the property γ also holds for the original interpretation $\mathcal{I}$.

Rule (R-G uproc): We consider $\mathcal{M} \models \psi \wedge \sigma$ and show $\mathcal{M} \models [P_1 \parallel P_2]\gamma$. Let $\mathcal{I} \in [\![P_1 \parallel P_2]\!]\mathcal{M}$ with $untime(\mathcal{I}) = \langle \mathcal{M}, a_0, \mathcal{M}_1, a_1, \dots, \mathcal{M}_n \rangle$. We show by induction over the length of $untime(\mathcal{I})$ that for all $\mathcal{M}_i$ with $i \geq 1$ one of the guarantee conditions, g_1 or g_2, holds.

$n = 1$: That is, $untime(\mathcal{I}) = \langle \mathcal{M}, a_0, \mathcal{M}_1 \rangle$. With $\mathcal{M} \models \psi$ and $\psi \vee g_2 \vdash r_1$ we can conclude that also $\mathcal{M} \models r_1$ holds and, symmetrically, $\mathcal{M} \models r_2$. So, if the

a_0-triggered, first transition of $\mathcal{I}$ is a transition of process P_1, we can apply the R-G condition (6.11) and get that $\mathcal{M}_1 \models g_1$. If it is a transition of P_2, analogously $\mathcal{M}_2 \models g_2$ holds.

$n = k+1$: So, $\mathit{untime}(\mathcal{I}) = \langle \mathcal{M}, a_0, \mathcal{M}_1, \ldots, \mathcal{M}_k, a_k, \mathcal{M}_{k+1} \rangle$. The induction hypothesis holds for $\langle \mathcal{M}, a_0, \ldots, \mathcal{M}_k \rangle$. Thus, without loss of generality $\mathcal{M}_k \models g_1$ (the other case is symmetric). If the a_k-triggered transition is a P_1 transition, then stability of g_1 yields $\mathcal{M}_{k+1} \models g_1$. Otherwise, it is a P_2 transition and we apply the premise of the rule $\psi \vee g_1 \vdash r_2$ to get $\mathcal{M}_k \models r_2$. With the R-G condition (6.11) for P_2, we conclude $\mathcal{M}_{k+1} \models g_2$. This proves that $\mathcal{M}_i \models g_1 \vee g_2$ for all $i \in 1..n$.

Thus, due to the right-most premise $g_1 \vee g_2 \models \overline{\gamma}$ the property γ holds for all $\mathcal{M}_i$ with $i \in 1..n$ (in case of $\overline{\gamma} = \sigma$ and $\gamma = \Box\sigma$) or for $\mathcal{M}_n$ (in case of $\overline{\gamma} = \gamma = \delta$). For $\gamma = \Box\sigma$ we additionally need the premise $\mathcal{M} \models \sigma$ to conclude $\mathcal{I} \models \Box\sigma$.

$\Box$

Checking R-G Conditions for dCSP and for DC

To apply rule (R-G uproc) in our context, the constraints over unknowns need to define R-G properties. Since these properties depend on the logic of the constraints, we exemplarily show how to specify R-G properties with DC formulae. In addition, proof rule (R-G uproc) can also be applied to CSP processes without unknown parts. In that case, the R-G property can be proven with the existing sequent calculus rules. We demonstrate these applications in the following.

DC R-G conditions. Preferably, a single DC formula is used to describe the R-G condition (6.11) in the sense that all interpretations of the DC formula satisfy the R-G condition. But DC formulae always describe complete system runs and not single transitions as in (6.11). Thus, a straightforward choice of an appropriate DC formula like

$$\neg\Diamond(\lceil r \vee g \rceil ^\frown \lceil \neg g \rceil)$$

does not fulfil the desired purpose, because it states that in every run of a concrete system component only allowed transitions occur—but it does not consider possible interfering state changes of parallel components.

Hence, to nevertheless verify R-G conditions with DC verification we use DC R-G formulae of the shape

$$\langle r, g \rangle_{\mathrm{DC}},$$

where r and g are DC state assertions. Formally, its semantics is given by the R-G condition (6.11).

A PEA $\mathcal{A}$ can be verified against DC R-G formulae $\langle r, g\rangle_{\mathrm{DC}}$ by proving with standard approaches, e.g., with DC model checking [Hoe06],

$$\mathcal{A}_{init} \models \neg(\lceil r \vee g\rceil ^\frown \lceil \neg g\rceil), \tag{6.12}$$

in which $\mathcal{A}_{init}$ equals $\mathcal{A}$ except that all locations are initial locations. With this small modification, we can check every single transition to satisfy the R-G condition.

Accordingly, a CSP-OZ-DC specification can be verified against a DC R-G formula $\langle r, g\rangle_{\mathrm{DC}}$—for instance when checking the constraints of a VA instantiated by a CSP-OZ-DC specification—by translating the specification into PEA and checking that the PEA satisfies $\langle r, g\rangle_{\mathrm{DC}}$ using the method described above.

A second solution for CSP-OZ-DC specifications is to separately verify the formulae in (6.12) for the OZ part using model checking. This yields the correctness of the R-G condition for all state changes, because the OZ part describes exactly[4] the state changes of the specification.

dCSP R-G conditions. For processes with data but without unknown parts the R-G condition can directly be proven within the sequent calculus of Sect. 4.2. Given such a process P with a set A of constrained events occurring in P and rely and guarantee formulae r and g, then R-G condition (6.11) can be verified by showing

$$\vdash \bigwedge_{a \bullet \varphi \in A} r \Rightarrow [a \bullet \varphi \rightarrow \mathtt{Skip}]g, \tag{6.13}$$

a property that can be proven in our calculus. It checks that every single event—and by this every state change—occurring in the process P satisfies the R-G condition.

So, using rule (R-G uproc) one can verify properties over parallel unknown parts and also over parallel processes without unknowns. The rule can also be used for verifying mixed compositions like

$$\Delta \vdash [(\mathtt{Proc}_{\backslash A,V} \bullet \langle r_1, g_1\rangle_{\mathrm{DC}}) \underset{B}{\parallel} P]\gamma,$$

where P satisfies R-G condition (6.13) for r_2 and g_2. P can also contain unknown parts as sub-processes—in this case, the unknown sub-processes must be of the shape $\mathtt{Proc}_{\backslash A,V} \bullet \langle r_2, g_2\rangle_{\mathrm{DC}}$.

6.1.4. Instantiating Parallel Unknowns

As explicated in Chap. 1, one of the basic ideas underlying the VA approach is that the processes instantiating unknown parts are separately verified against the assumptions on the unknown parts in order to get a correct instantiation of a VA. We wish to

[4]This is only valid if the DC part does not restrict the state changes of the specification. Otherwise, checking the OZ part alone is not a complete approach to verify the R-G condition.

instantiate parallel unknown processes by concrete processes in a compositional way: given a process

$$\mathtt{Proc}_{\backslash A_1, V_1} \bullet F_1 \parallel_B \mathtt{Proc}_{\backslash A_2, V_2} \bullet F_2$$

satisfying property γ, then for all instantiations P_i of $\mathtt{Proc}_{\backslash A_i, V_i}$ that satisfy F_i, for $i \in 1..2$, the process $P_1 \parallel_B P_2$ shall also satisfy γ. But Example 6.1.9 shows that this fails even for simple cases. As elucidated in the previous section, the problems are caused by instantiations of unknown parts that do introduce interferences on shared variables.

Instantiation for R-G assumptions. If the properties over the unknowns are R-G conditions according to the previous section, then the instantiating processes cannot introduce interferences by construction of the R-G constraints. The argument is that if the correctness of a process has been proven with rule (R-G uproc), one essentially has shown that every process satisfying the R-G condition is correct with respect to the desired property. This is reflected by the fact that the rule is actually independent of P_1 and P_2 except for the side-condition that the R-G condition holds. Thus, the desired property established by the application of the rule also holds for all possible instances P_1 and P_2 satisfying the assumptions, which are in the form of R-G conditions.

Instantiation for the general case. If the assumption on unknowns are general real-time properties, and the correctness of a VA process has been proven, e.g., by the translation-based approach of Sect. 6.1.2, then some extra effort is necessary to ensure that the refining processes do not introduce interferences violating a desired property.

One possible approach is to show that every state change of an instantiating process does not influence a parallel process in a property-violating way. But we follow the line of the refinement notion from Chap. 5.1 and provide some syntactic conditions under which interference-freedom is guaranteed. In this manner, we can keep up the idea that the instantiation of a VA is verified in two steps: with an efficient matching check for the structure of the architecture and the automatic verification of the assumptions.

Theorem 6.1.11 (Interference-freedom for parallel processes)
Given a process $P_1 \parallel_A P_2$, where P_1 and P_2 satisfy safety properties $P_1 \models F_1$ and $P_2 \models F_2$, then these properties are also valid for the parallel composition

$$P_1 \parallel_A P_2 \models F_1 \wedge F_2$$

if for all variables and events x in P_1 and P_2 one of the following conditions holds:

1. *x is local, i.e., it is read and changed by only one of P_1 or P_2*

2. *x is changed synchronously, i.e., P_1 and P_2 agree on all state changes of x*
3. *x has no influence on F_i, i.e., x is changed in P_1 and $P_2 \models F_2$ for all possible changes of x or, vice versa, x is changed in P_2 and $P_1 \models F_1$ for all possible changes of x.*

The first two conditions are actually syntactical conditions that can be checked by inspecting every constraint of P_1 and P_2. The third condition can be verified by composing P_i in parallel to a chaos process that arbitrarily changes x with every state change and checking $P_i \models F_i$ for this composition.

Proof. The proof is by contradiction, that is, postulating that the preconditions of the theorem are valid, we assume that there is an interpretation

$$\mathcal{I} \in [\![P_1 \underset{A}{\parallel} P_2]\!] \text{ with } \mathcal{I} \notin [\![F_1 \wedge F_2]\!].$$

Without loss of generality, we particularly assume $\mathcal{I} \notin [\![F_1]\!]$. Since F_1 only contains symbols from the alphabet of P_1, the same holds for the restriction of $\mathcal{I}$ to symbols in P_1, written $\mathcal{I}_{|P_1}$:

$$\mathcal{I}_{|P_1} \notin [\![F_1]\!]. \tag{6.14}$$

Let P_1^* be the process that behaves like P_1 except that arbitrary state changes for variables of property 3. are possible. Hence, property 3. is wrt. P_1 equivalent to $P_1^* \models F_1$. We show by induction over the length of *untime*($\mathcal{I}$) that the restriction of $\mathcal{I}$ to symbols in P_1 is also an interpretation of P_1^*, i.e., $\mathcal{I}_{|P_1} \in [\![P_1^*]\!]$.

For the base case $\mathcal{I} = \langle \mathcal{M} \rangle$, $\mathcal{I}_{|P_1}$ is trivially an interpretation of P_1^*. Let $\mathcal{I} = \overline{\mathcal{I}} \frown \langle \mathcal{M}, a, \mathcal{N} \rangle$, where $\overline{\mathcal{I}}_{|P_1}$ is already an interpretation of P_1^*. The state change $\langle \mathcal{M}, a, \mathcal{N} \rangle$, say of a variable x, can be triggered by P_1, P_2, or by a synchronous transition. Moreover, one of the conditions 1. up to 3. holds for x by precondition:

1. x is local to P_1 and, thus, cannot be changed by P_2. P_1^* can execute the same state change as P_1.
2. x is changed synchronously by P_1 and P_2. Hence, according to Def. 6.1.1 there is a corresponding transition with guard $\updownarrow a \wedge \varphi_1 \wedge \varphi_2$ with $\mathcal{N}_1 \cup \mathcal{N}_2' \models \updownarrow a \wedge \varphi_1 \wedge \varphi_2$ in the PEA representing the semantics of $P_1 \underset{A}{\parallel} P_2$. This is only possible if there is a corresponding transition of P_1 with guard $\updownarrow a \wedge \varphi_1$. The transition can be taken, because of $\mathcal{N}_1 \cup \mathcal{N}_2' \models \updownarrow a \wedge \varphi_1$ by which $\mathcal{I}_{|P_1}$ is also a possible P_1^* interpretation.
3. x satisfies condition 3. Every state change of x can also be performed by P_1^* due to the definition of condition 3: in P_1^* the same transitions and state changes as in P_1 are possible and, in the case that the state change in $\mathcal{I}$ has been triggered by a P_2 transition, P_1^* may arbitrarily change x.

Thus, $\mathcal{I}_{|P_1} \in [\![P_1^*]\!]$ and with condition 3. we can conclude $\mathcal{I}_{|P_1} \in [\![F_1]\!]$, which contradicts the assumption (6.14). ☐

6.1.5. An Interpretation-Based Semantics for Parallel Unknowns

In Sect. 3.2, we noticed that the interpretation-based semantics of Sect. 3.1.1 is not suited to describe parallelism over constrained unknown processes. Thus, we have introduced a PEA-based semantics at the beginning of this chapter but without relating it to the original semantics. We make up for it in this section by suggesting a semantics for parallel unknowns that fits to the original interpretation-based semantics and that is equivalent to the PEA-semantics.

Interfered interpretation semantics. As we have seen in Sect. 3.2, the problem with the pure interpretation semantics for parallel processes is that the interpretations do not reflect behaviour that can only occur in a parallel composition and not in the isolated execution of the process. One can think of a path in the execution that is disabled by default and may be enabled in a parallel context like in Example 3.2.2. Such "disabled" paths are not visible in the interpretation semantics but nevertheless important when calculating the semantics of parallel compositions. Thus, what we need is a semantics reflecting such potential behaviour of a process that may be enabled in a parallel composition.

For this reason, we extend the interpretation semantics by interpretations containing all possible *interference intervals*. An *interfered interpretation* is, similarly to a standard interpretation, a mapping $\mathcal{I} : \mathit{Time} \nrightarrow \mathit{Model}$, but now $\mathit{untime}(\mathcal{I})$ is of the shape

$$\mathit{untime}(\mathcal{I}) = \langle \mathcal{M}_0, \tau, \overline{\mathcal{M}}_0, a_1, \mathcal{M}_1, \tau, \overline{\mathcal{M}}_1, a_2, \mathcal{M}_2, \dots, \mathcal{M}_{n-1}, a_n, \mathcal{M}_n \rangle,$$

in which τ symbolically represents an interference interval, i.e., $\mathcal{I}(t)$ is undefined in these intervals, which is modelled by the empty model that is undefined for every symbol. The idea is that the interpretation may be completed with an interfering parallel component. More precisely, there are points in time $t_i^a, t_i^u, t_i^e \in \mathit{Time}$ for $i \in 0..n$ such that

$$0 = t_0^a < t_0^u \leq t_0^e < t_1^a < t_1^u \leq t_1^e < t_2^a < \dots$$

with

$$\begin{aligned} \mathcal{I}(t) &= \mathcal{M}_j && \text{for } t_j^a \leq t < t_j^u \\ \mathcal{I}(t) &= \text{undefined} && \text{for } t_j^u \leq t < t_j^e \\ \mathcal{I}(t) &= \overline{\mathcal{M}}_j && \text{for } t_j^e \leq t < t_{j+1}^a \end{aligned}$$

for $j \in 0..n-1$. So, an interfered interpretation where the undefined parts have a length of zero and $\mathcal{M}_i = \overline{\mathcal{M}}_i$ is exactly a standard interpretation.

Such an interfered interpretation fits to a run $\pi = \langle a_1 \bullet \varphi_1, a_2 \bullet \varphi_2, \dots \rangle$ of the LTS of a process P if for all $i \geq 0$

$$\overline{\mathcal{M}}_i(a_{i+1}) \neq \mathcal{M}_{i+1}(a_{i+1}).$$

The $\mathcal{I}$ is in the interfered semantics of P, denoted $\mathcal{I} \in [\![P]\!]^i \mathcal{M}_0$, if the conditions from Sect. 3.1.1 hold analogously for models $\overline{\mathcal{M}}_i$ and $\mathcal{M}_{i+1}$; but additionally at all interference intervals, marked with τ, arbitrary state changes are possible:

- $\mathcal{I}$ fits to a run π of the LTS of P
- $(\overline{\mathcal{M}}_{i-1} \cup \mathcal{M}_i) \models \varphi_i$ for $i > 0$
- constant symbols are not changed: $\mathcal{M}_i(v) = \overline{\mathcal{M}}_i(v) = \mathcal{M}_{i+1}(v)$ for $v \in \mathit{Const}$
- for $a_n = \checkmark$ no symbol is changed: $\overline{\mathcal{M}}_{n-1}(v) = \mathcal{M}_n(v)$, $v \in \mathit{SysVar}$.

Example 6.1.12. Given two processes

$$
\begin{aligned}
P &\stackrel{c}{=} a \bullet \varphi \rightarrow \texttt{Stop} \\
Q &\stackrel{c}{=} b \bullet \psi \rightarrow \texttt{Stop}
\end{aligned}
$$

with corresponding untimed interpretations

$$
\begin{aligned}
\mathcal{I}_P &= \langle \mathcal{M}_1, \tau, \mathcal{M}_2, a, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \\
\mathcal{I}_Q &= \langle \mathcal{N}_1, \tau, \mathcal{N}_2, b, \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle,
\end{aligned}
$$

where $\mathcal{M}_1 = \mathcal{N}_1$ and $\mathcal{M}_4 = \mathcal{N}_4$, then the merged interpretation

$$
\langle \mathcal{M}_1, \tau, \mathcal{N}_2, b, \mathcal{N}_3, \tau, \mathcal{M}_2, a, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \tag{6.15}
$$

is in the semantics of the parallel composition of the processes, $\mathcal{I}_P \parallel \mathcal{P}_Q \in [\![P \parallel Q]\!]^i \mathcal{M}_0$. Even though we only have considered untimed interpretations here, $\mathcal{I}_P$ and $\mathcal{I}_Q$ must also agree on their timing behaviour in the sense that for every point in time t

$$
(\mathcal{I}_P \parallel \mathcal{I}_Q)(t) = \begin{cases} \mathcal{I}_P(t) & \text{if } \mathcal{I}_Q(t) = \text{undefined} \\ \mathcal{I}_Q(t) & \text{if } \mathcal{I}_P(t) = \text{undefined} \\ \text{undefined} & \text{if } \mathcal{I}_P(t) = \text{undefined and } \mathcal{I}_Q(t) = \text{undefined} \\ \mathcal{I}_P(t) = \mathcal{I}_Q(t) & \text{otherwise.} \end{cases} \tag{6.16}
$$

Correlation with interpretation semantics. The interfered semantics of a process, $[\![P]\!]^i \mathcal{M}$, contains all runs of P with its corresponding data changes for every possible interference of another parallel process. In particular, it contains as a subset the standard interpretations of P:

$$
[\![P]\!]\mathcal{M} \subset [\![P]\!]^i \mathcal{M},
$$

namely the interpretations without interference, i.e., an interpretation

$$\langle \mathcal{M}_0, \tau, \overline{\mathcal{M}}_0, a_1, \mathcal{M}_1, \tau, \overline{\mathcal{M}}_1, a, \dots \rangle$$

without undefined parts and $\mathcal{M}_i = \overline{\mathcal{M}}_i$ for all i. In particular,

$$[\![P]\!]\mathcal{M} = \{\mathcal{I} \mid \mathcal{I} \in [\![P]\!]^i\mathcal{M} \text{ and } \mathcal{I} = \langle \mathcal{M}, a, \mathcal{M}_1, b, \mathcal{M}_2, \dots \rangle\}. \tag{6.17}$$

In contrast to the semantics of Sect. 3.1.1, we can now express the parallel composition operator in terms of the semantics of its constituents. The definition is analogous to the definition of the parallel composition over interpretations for non-interfering processes on page 54. For the sake of simplicity, we omit the timing part in the following definition and instead demand additionally that condition (6.16) holds in each case. Furthermore, we write $\langle \mathcal{M}, \tau, \mathcal{N}, a \rangle \frown \mathcal{I}$ as abbreviation for the interpretation $\langle \mathcal{M}, \tau, \mathcal{N}, a, \overline{\mathcal{M}}, \tau, \overline{\mathcal{N}}, b, \dots \rangle$ if $\mathcal{I} = \langle \overline{\mathcal{M}}, \tau, \overline{\mathcal{N}}, b, \dots \rangle$.

The parallel composition of interfered interpretations, $\mathcal{I}_1 \parallel_A \mathcal{I}_2$, is for $a, a_1, a_2 \in A$ with $a_1 \neq a_2$ and $b, b_1, b_2 \notin A$ defined by

Commutativity:

$$\mathcal{I}_1 \parallel_A \mathcal{I}_2 = \mathcal{I}_2 \parallel_A \mathcal{I}_1$$

No step:

$$\langle \mathcal{M}, \tau, \mathcal{N} \rangle \parallel_A \langle \mathcal{M}, \tau, \mathcal{N} \rangle = \{\langle \mathcal{M}, \tau, \mathcal{N} \rangle\}$$
$$\langle \mathcal{M}, \tau, \mathcal{N} \rangle \parallel_A \langle \mathcal{M}, \tau, \mathcal{N}, a, \overline{\mathcal{M}}, \tau, \overline{\mathcal{N}} \rangle = \{\langle \mathcal{M}, \tau, \mathcal{N} \rangle\}$$

Single step:

$$\langle \mathcal{M}, \tau, \overline{\mathcal{N}} \rangle \parallel_A \langle \mathcal{M}, \tau, \overline{\mathcal{M}}, b, \mathcal{N}, \tau, \overline{\mathcal{N}} \rangle = \{\langle \mathcal{M}, \tau, \overline{\mathcal{M}}, b, \mathcal{N}, \tau, \overline{\mathcal{N}} \rangle\}$$

Single step (waiting a):

$$\langle \mathcal{M}, \tau, \mathcal{N}, a \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{M}, \tau, \mathcal{N}_1, b, \mathcal{N}_2, \tau, \mathcal{N}_3 \rangle \frown \mathcal{I}_2 =$$
$$\{\langle \mathcal{M}, \tau, \mathcal{N}_1, b \rangle \frown \mathcal{I} \mid \mathcal{I} \in \langle \mathcal{N}_2, \tau, \mathcal{N}, a \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{N}_2, \tau, \mathcal{N}_3 \rangle \frown \mathcal{I}_2\}$$

Synchronous step:

$$\langle \mathcal{M}, \tau, \mathcal{N}, a \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{M}, \tau, \mathcal{N}, a \rangle \frown \mathcal{I}_2 =$$
$$\{\langle \mathcal{M}, \tau, \mathcal{N}, a \rangle \frown \mathcal{I} \mid \mathcal{I} \in \mathcal{I}_1 \parallel_A \mathcal{I}_2\}$$

Blocking:

$$\langle \mathcal{M}, \tau, \mathcal{N}, a_1 \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{M}, \tau, \mathcal{N}, a_2 \rangle \frown \mathcal{I}_2 = \{\langle \mathcal{M}, \tau, \mathcal{N} \rangle\}$$

Asynchronous steps:

$$\begin{aligned}
&\langle \mathcal{M}_1, \tau, \mathcal{M}_2, b_1, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{N}_1, \tau, \mathcal{N}_2, b_2, \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle \frown \mathcal{I}_2 = \\
&\{\langle \mathcal{M}_1, \tau, \mathcal{M}_2, b_1 \rangle \frown \mathcal{I} \mid \mathcal{M}_1 = \mathcal{N}_1 \wedge \\
&\qquad \mathcal{I} \in \langle \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{M}_3, \tau, \mathcal{N}_2, b_2, \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle \frown \mathcal{I}_2\} \cup \\
&\{\langle \mathcal{N}_1, \tau, \mathcal{N}_2, b_2 \rangle \frown \mathcal{I} \mid \mathcal{M}_1 = \mathcal{N}_1 \wedge \\
&\qquad \mathcal{I} \in \langle \mathcal{N}_3, \tau, \mathcal{M}_2, b_1, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \frown \mathcal{I}_1 \parallel_A \langle \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle \frown \mathcal{I}_2\}.
\end{aligned}$$

Remark 6.1.13. In addition, we demand that for every occurrence $\mathcal{M}, b, \mathcal{N}$ of an event $b \notin A$ the constraint $\mathcal{M} \cup \mathcal{N}' \models \bigwedge_{a \in A} \boxminus a$ is valid, which basically states that actually no event a occurs simultaneously with b. This is necessary, because the occurrence of an event $a \in A$ must always enforce synchronisation.

The definition basically reflects that if two interfered interpretations are composed in parallel the next step is always either a synchronous step (i.e., both interpretations need to agree for this step) or an asynchronous step of one component (in which the state change of the step is executed during an explicit inference phase marked by τ of the other component). The remaining process operators on interfered interpretations are defined like the operators for standard interpretations in Sect. 3.2.1 except for the necessary syntactic extension to include the interference intervals in the interpretations and except for unknown processes, for which the semantics is defined in the next section.

Example 6.1.14. We continue Example 6.1.12 and show that (6.15) actually is in the semantics of $\mathcal{I}_P \parallel \mathcal{I}_Q$.

$$\begin{aligned}
\mathcal{I}_P \parallel \mathcal{I}_Q = &\{\langle \mathcal{M}_1, \tau, \mathcal{M}_2, a \rangle \frown \mathcal{I} \mid \mathcal{M}_1 = \mathcal{N}_1 \wedge \\
&\quad \mathcal{I} \in \langle \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \parallel \langle \mathcal{M}_3, \mathcal{N}_2, b, \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle\} \cup \\
&\{\langle \mathcal{N}_1, \tau, \mathcal{N}_2, b \rangle \frown \mathcal{I} \mid \mathcal{M}_1 = \mathcal{N}_1 \wedge \\
&\quad \mathcal{I} \in \langle \mathcal{N}_3, \tau, \mathcal{M}_2, a, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle \parallel \langle \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle\} \\
= &\{\langle \mathcal{M}_1, \tau, \mathcal{M}_2, a, \mathcal{M}_3, \tau, \mathcal{N}_2, b, \mathcal{N}_3, \tau, \mathcal{N}_4 \rangle\} \cup \\
&\{\langle \mathcal{N}_1, \tau, \mathcal{N}_2, b, \mathcal{N}_3, \tau, \mathcal{M}_2, a, \mathcal{M}_3, \tau, \mathcal{M}_4 \rangle\}
\end{aligned}$$

For the first equality, the $\parallel$-definition for the asynchronous step is applied. Because of $\mathcal{M}_4 = \mathcal{N}_4$ the $\parallel$-definition for the single step can be applied to each of the sets.

Interfered semantics for unknowns. For a constrained unknown process $\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F$, we assume that F can be represented by a set of interfered interpretations $[\![F]\!]^i$. The semantics of $\texttt{Proc}^{(\infty)}_{\setminus A,V} \bullet F$ can then be defined by interfered interpretations

$$[\![\texttt{Proc}_{\setminus A,V} \bullet F]\!]^i := [\![F]\!]^i \underset{alph(F)}{\|} \{\mathcal{I} \mid untime(\mathcal{I}) = \langle \mathcal{M}_0, \tau, \overline{\mathcal{M}}_0, a_1, \mathcal{M}_1, \tau, \overline{\mathcal{M}}_1, \dots \rangle, \text{ where } a_i \notin A, \mathcal{M}_i \cup \overline{\mathcal{M}}_{i+1} \models v' = v\},$$

i.e., it contains arbitrary interfered interpretations as long as the exclusion sets A and V are respected, and F is respected for all symbols from the alphabet of F. In case of the infinite unknown process $\texttt{Proc}^{\infty}$, we require that the interpretations in the set above do not contain the $\checkmark$-event.

Correlation with PEA semantics. We justify that the interfered interpretation semantics actually coincides with the PEA-based semantics that is used in Sect. 6.1.2 to define a translation of processes into timed automata. To this end, we argue that the relationship in (6.7), i.e.,

$$\mathcal{I} \in [\![P]\!] \text{ iff there is a run } \pi \text{ of } PEA_S(P) \text{ matching } \mathcal{I},$$

is true if we use the interfered interpretation semantics as basis for the semantics of $[\![P]\!]$. So, let $[\![P]\!]$ be now defined by the interfered interpretation semantics as in equation (6.17).

We state in the following corollary that the interpretations of a process P are exactly the interpretations that are matched by $PEA_S(P)$.

Corollary 6.1.15 (Equivalence of processes and PEA)
If P is a process with data and unknown processes in guarded normal form, then

$$\mathcal{I} \in [\![P]\!] \textit{ iff there is a run } \pi \textit{ of } PEA_S(P) \textit{ matching } \mathcal{I}.$$

Instead of proving this corollary directly, we show the stronger result that the interfered interpretations of P and PEA_S coincide. To this end, we add interference intervals to PEA configurations with the same idea as for interfered interpretations of processes. Accordingly, runs of PEA are straightforwardly extended to runs of such interfered configurations, and an interfered run matches interfered interpretations analogously to the definition of the interfered interpretations for a process on page 150. We denote the set of interfered interpretations that are matched by runs of an automaton $\mathcal{A}$ by $[\![\mathcal{A}]\!]^i$.

The following lemma shows that the asynchronous parallel composition on PEA can be expressed by the interfered interpretations of its constituents.

Lemma 6.1.16
For PEA $\mathcal{A}_i = (P_i, V_i, A_i, C_i, E_i, s_i, I_i, P_i^0)$, $i \in 1..2$, the following equality holds

$$[\![\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2]\!]^i = [\![\mathcal{A}_1]\!]^i \underset{B}{\|} [\![\mathcal{A}_2]\!]^i.$$

Proof. We show by induction over the length of $untime(\mathcal{I})$ that for every interpretation of $[\![\mathcal{A}_1 \tilde{\|}_B \mathcal{A}_2]\!]^i$ that corresponds to a run π ending in a location (p_1, p_2), there are interpretations $\mathcal{I}_1 \in [\![\mathcal{A}_1]\!]^i$ and $\mathcal{I}_2 \in [\![\mathcal{A}_2]\!]^i$ with $\mathcal{I} = \mathcal{I}_1 \|_B \mathcal{I}_2$, corresponding to runs ending in the locations p_1 and p_2, respectively. And the same holds for the other direction.

For the base case $untime(\mathcal{I}) = \langle \mathcal{M}, \tau, \mathcal{N} \rangle$, $\mathcal{I} \in [\![\mathcal{A}_1 \tilde{\|}_B \mathcal{A}_2]\!]^i$ is equivalent to $\mathcal{I} \in [\![\mathcal{A}_1]\!]^i \|_B [\![\mathcal{A}_2]\!]^i$ with $\mathcal{I} \in \mathcal{I}_1 \|_B \mathcal{I}_2$, and the runs matching $\mathcal{I}, \mathcal{I}_1$ and $\mathcal{I}_2$ end up in the start locations of the automata, (p_1^0, p_2^0) and p_1^0 and p_2^0, respectively.

So, we now consider an interpretation

$$\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \quad \text{with } \mathcal{I} = \overline{\mathcal{I}} \frown \langle \mathcal{M}_1, \tau, \mathcal{M}_2 \rangle. \tag{6.18}$$

From the induction hypothesis we conclude $\mathcal{I} \in [\![\mathcal{A}_1 \tilde{\|}_B \mathcal{A}_2]\!]^i$ iff $\mathcal{I} \in [\![\mathcal{A}_1]\!]^i \|_B [\![\mathcal{A}_2]\!]^i$ such that $\mathcal{I} = \mathcal{I}_1 \|_B \mathcal{I}_2$, and $\mathcal{I}$ is matched by a run π of $\mathcal{A}_1 \tilde{\|}_B \mathcal{A}_2$ ending up in a location (p, q). Moreover, according to the induction hypothesis we assume that π_1 matches $\mathcal{I}_1$ and π_2 matches $\mathcal{I}_2$ such that π_1 ends up in location p and π_2 in q. By definition of $\|$ both, $\mathcal{I}_1$ and $\mathcal{I}_2$, also end with the model $\mathcal{M}_2$ like this is the case for $\mathcal{I}$.

We examine the next transition step of the automata causing the a-transition of (6.18).

- Asynchronous step ($a \notin B$): We assume $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in [\![\mathcal{A}_1 \tilde{\|}_B \mathcal{A}_2]\!]^i$. That is, there is by definition of $\tilde{\|}_B$ (without loss of generality) a transition

  $$((p, q), \bigwedge_{b \in B} \boxminus\, b \wedge \varphi, X, (p', q))$$

  with $\mathcal{M}_2 \cup \mathcal{N}_1' \models \bigwedge_{b \in B} \boxminus\, b \wedge \varphi$. The timing constraints that are possibly contained in φ are also not violated by the state invariant of p'. Hence, due to Def. 6.1.1 there also is a transition (p, φ, X, p'). Thus, since $\mathcal{I}_1$ is an interpretation of $\mathcal{A}_1$ ending with the model $\mathcal{M}_2$,

  $$\mathcal{I}_1 \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \tag{6.19}$$

  is also an interpretation of $\mathcal{A}_1$. The interpretation $\mathcal{I}_2$ is of the shape $\mathcal{I}_2 = \overline{\mathcal{I}_2} \frown \langle \overline{\mathcal{M}}_1, \tau, \mathcal{M}_2 \rangle$ for a particular $\overline{\mathcal{M}}_1$. Due to the definition of $\|_B$, this model $\overline{\mathcal{M}}_1$ necessarily occurs somewhere in $\mathcal{I}_1$, because there exists an $\mathcal{I}$ with $\mathcal{I} \in \mathcal{I}_1 \|_B \mathcal{I}_2$. Since in $[\![\mathcal{A}_2]\!]^i$ arbitrary interferences are contained, we can extend $\mathcal{I}_2$ to

  $$\overline{\mathcal{I}}_2 \frown \langle \overline{\mathcal{M}}_1, \tau, \mathcal{N}_2 \rangle. \tag{6.20}$$

This extension does not violate any timing constraints (i.e., clock invariants on q), because $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle$ is an interpretation of $\mathcal{A}_1 \tilde{\|} \mathcal{A}_2$ as stated in (6.18).

By definition of the parallel composition of the interfered interpretations in (6.19) and (6.20):

$$\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in \big(\mathcal{I}_1 \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \underset{B}{\|} \overline{\mathcal{I}}_2 \frown \langle \overline{\mathcal{M}}_1, \tau, \mathcal{N}_2 \rangle\big),$$

because $\overline{\mathcal{M}}_1$ is contained in $\mathcal{I}_1$ and $\mathcal{I} \in \mathcal{I}_1 \underset{B}{\|} \mathcal{I}_2$. With this, $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in \llbracket \mathcal{A}_1 \rrbracket^i \underset{B}{\tilde{\|}} \llbracket \mathcal{A}_2 \rrbracket^i$.

Vice versa, the argument is similar: if $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle$ is an interpretation of $\llbracket \mathcal{A}_1 \rrbracket^i \underset{B}{\tilde{\|}} \llbracket \mathcal{A}_2 \rrbracket^i$, we need to show that it is also an interpretation of $\llbracket \mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2 \rrbracket^i$. Without loss of generality, the last transition of (6.18) is triggered by a transition $(p, \updownarrow a \wedge \varphi, X, p')$ with $\mathcal{M}_2 \cup \mathcal{N}_1' \models \updownarrow a \wedge \varphi$. Due to the condition from Remark 6.1.13, we also know $\mathcal{M}_2 \cup \mathcal{N}_1' \models \bigwedge_{b \in B} \boxminus b$. Hence, the parallel composition $\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2$ has a transition

$$((p, q), \bigwedge_{b \in B} \boxminus b \wedge \updownarrow a \wedge \varphi, X, (p', q)). \tag{6.21}$$

Due to the induction hypothesis we know that $\mathcal{I}$ (in (6.18)) is matched by a run of $\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2$ that ends in location (p, q). Since $\mathcal{M}_2 \cup \mathcal{N}_1' \models \updownarrow a \wedge \varphi \wedge \bigwedge_{b \in B} \boxminus b$ transition (6.21) can be taken with $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle$ as corresponding interpretation. The timing behaviour cannot prevent the transition, because then this would be also the case for one of the single automata $\mathcal{A}_1$ or $\mathcal{A}_2$. Thus, $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in \llbracket \mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2 \rrbracket^i$.

- Synchronous step ($a \in B$): For the interpretation $\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle$ from (6.18), there must be a transition

$$((p, q), \updownarrow a \wedge \varphi_1 \wedge \varphi_2, X_1 \cup X_2, (p', q'))$$

of $\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2$ and corresponding transitions

$$(p, \updownarrow a \wedge \varphi_1, X_1, p')$$
$$(q, \updownarrow a \wedge \varphi_2, X_2, q')$$

of $\mathcal{A}_1$ and $\mathcal{A}_2$. For these transitions, we can construct the interpretations

$$\mathcal{I}_1 \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in \llbracket \mathcal{A}_1 \rrbracket^i$$
$$\mathcal{I}_2 \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in \llbracket \mathcal{A}_2 \rrbracket^i,$$

and conclude by definition of $\|$

$$\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in [\![\mathcal{A}_1]\!]^i \underset{B}{\|} [\![\mathcal{A}_2]\!]^i,$$

which is identical to the interpretation of the product

$$\mathcal{I} \frown \langle a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \in [\![\mathcal{A}_1 \underset{B}{\tilde{\|}} \mathcal{A}_2]\!]^i.$$

□

Theorem 6.1.17 (Interfered interpretations of processes and PEA)
If P is a process with data and unknown processes in guarded normal form, then

$$\mathcal{I} \in [\![P]\!]^i \quad \textit{iff} \quad \mathcal{I} \in [\![PEA_S(P)]\!]^i.$$

Proof. We show by induction over the structure of P that the interfered interpretations $\mathcal{I}$ of $[\![P]\!]$ coincide with the interfered interpretations matched by runs of $PEA_S(P)$. We show the most important cases:

$PEA_S(\texttt{Stop})$**:** The PEA for `Stop` accepts only interpretations $\langle \mathcal{M}, \tau, \mathcal{N} \rangle$, which are exactly the interpretations of `Stop`.

$PEA_S(a \bullet \varphi \to P)$**:** The PEA consists of a start location allowing an a-transition to a second location representing P. Thus, the PEA has the interfered interpretations

$$\langle \mathcal{M}_1, \tau, \mathcal{M}_2, a, \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \frown \mathcal{I}, \tag{6.22}$$

in which $\langle \mathcal{N}_1, \tau, \mathcal{N}_2 \rangle \frown \mathcal{I} \in [\![P]\!]^i$ by induction hypothesis. Since $\mathcal{M}_2 \cup \mathcal{N}_1' \models \varphi$ for the interpretation of the automaton and also for the process interpretation, the interpretation (6.22) is also in $[\![a \to P]\!]^i$. Conversely, when assuming that (6.22) is in $[\![a \to P]\!]^i$, we can conclude in the same way that it is also an interpretation of $PEA_S(a \bullet \varphi \to P)$.

$PEA_{\texttt{Proc}}(\texttt{Proc}^{(\infty)}_{\backslash A, V} \bullet F)$**:** The semantics of $\texttt{Proc}^{(\infty)}_{\backslash A, V} \bullet F$ is defined by

$$[\![F]\!]^i \underset{alph(F)}{\|} [\![\texttt{Proc}^{(\infty)}_{\backslash A, V}]\!]^i,$$

in which

$$[\![\texttt{Proc}^{(\infty)}_{\backslash A, V}]\!]^i := \{\mathcal{I} \mid untime(\mathcal{I}) = \langle \mathcal{M}_0, \tau, \overline{\mathcal{M}}_0, a_1, \mathcal{M}_1, \tau, \overline{\mathcal{M}}_1, \ldots \rangle,$$
$$\text{where } a_i \notin A, \mathcal{M}_i \cup \overline{\mathcal{M}}_{i+1} \models v' = v\},$$

and $PEA_{\texttt{Proc}}$ is pictured in Fig. 6.1. It holds that

$$[\![PEA(F)]\!]^i = [\![F]\!]^i \tag{6.23}$$

because of the requirement that $PEA(F)$ exactly represents the semantics of F. The set $[\![\texttt{Proc}^{(\infty)}_{\backslash A,V}]\!]^i$ is the interference semantics of the left PEA of Fig. 6.1 (we denote it $PEA(\texttt{Proc})$ here)

$$[\![\texttt{Proc}^{(\infty)}_{\backslash A,V}]\!]^i = [\![PEA(\texttt{Proc})]\!]^i, \tag{6.24}$$

which can be seen by comparing the possible transitions: in both $[\![\texttt{Proc}^{(\infty)}_{\backslash A,V}]\!]^i$ and the PEA any event is possible except for events from A and any state change may performed, as long as the symbols from V are not changed. In case of the infinite unknown process, no ✓-event is performed.

Thus, with application of Lemma 6.1.16, we conclude

$$\begin{aligned}
[\![PEA_{\texttt{Proc}}(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F)]\!]^i &= [\![PEA(\texttt{Proc}) \underset{alph(F)}{\tilde{\|}} PEA(F)]\!]^i \\
\{\text{Lemma 6.1.16}\} &= [\![PEA(\texttt{Proc})]\!]^i \underset{alph(F)}{\|} [\![PEA(F)]\!]^i \\
\{(6.23),(6.24)\} &= [\![\texttt{Proc}^{(\infty)}_{\backslash A,V}]\!]^i \underset{alph(F)}{\|} [\![F]\!]^i \\
\{\text{Def.}\} &= [\![\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F]\!]^i.
\end{aligned}$$

$PEA_S(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F \underset{B}{\tilde{\|}} Q)$**:** We can again apply Lemma 6.1.16 (for the second equality) and the induction hypothesis (for the third equality):

$$\begin{aligned}
[\![PEA_S(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F \underset{B}{\tilde{\|}} Q)]\!]^i &= [\![PEA_S(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F) \underset{B}{\tilde{\|}} PEA_S(Q)]\!]^i \\
&= [\![PEA_S(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F)]\!]^i \underset{B}{\|} [\![PEA(Q)]\!]^i \\
&= [\![\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F]\!]^i \underset{B}{\|} [\![Q]\!]^i.
\end{aligned}$$

$PEA_S(\texttt{Proc}^{(\infty)}_{\backslash A,V} \bullet F \fatsemi Q)$**:** The proposition holds for this case, because sequential composition on PEA is a congruence for the interfered trace semantics: i.e., it generally holds that

$$[\![\mathcal{A}_1 \fatsemi \mathcal{A}_2]\!]^i = [\![\mathcal{A}_1]\!]^i \fatsemi [\![\mathcal{A}_2]\!]^i.$$

Thus, we can use the same argument as in the previous case.

□

Corollary 6.1.15 is a direct conclusion from this result. Hence, this shows that the PEA semantics, that is used for the region construction in the context of proof rule (parallel uproc), is actually a sound representation of processes, because the PEA semantics and the interfered interpretation semantics for processes with parallelism over unknowns are equivalent. For processes without parallelism over unknowns, both of these semantics comply with the standard interpretation semantics from Sect. 3.1.1.

6.2. Verifying Timing Properties

In eCSP timing properties are integrated by imposing real-time assumptions on unknown processes. Using the calculus from Chap. 4, it is possible to verify validity of dCSP formulae over eCSP processes. With the box and the diamond operator of dCSP we can specify temporal propositions to state that properties are valid somewhere on the execution or always during the execution of a process. So with dCSP, untimed safety properties are verified that may depend on timed processes. Until now, it is not possible to directly prove real-time properties over processes with dCSP. Verification of untimed safety properties predominantly suffices in practice, because safety of an entire system usually means the exclusion of some unwanted event, like collision freedom, shared access to a critical resource, or high pressure in a tank etc. (cf. the case studies in Chap. 8). These events are described with untimed properties, even though they are dependent on timed behaviour.

However, in some cases it can be more natural to define safety in terms of timed properties. Thus, it is also an interesting question how real-time properties can be proven with dCSP and our sequent calculus.

Basically there are two approaches to integrate timing properties into dCSP. One possibility is to introduce new proof rules for the handling of real-time formulae. This violates our requirement that the calculus is independent of the logic in use, but it allows us to reason about timed formulae uniformly in our calculus. On the other hand, we can use a translation or automata-theoretic approach [VW86, KVW00, DL02], i.e., the negation of the real-time property to be verified is translated into a process, and we then verify that the parallel composition of the original process and the property process is empty.

Even though these approaches are not in the focus of this work, we briefly sketch both of them in the following sections. To this end, the syntax of dCSP is extended to include timed properties. That is, we also allow formulae

$$[P]F \text{ and } \langle P\rangle F,$$

where P is an eCSP process and F a timed formulae with a semantics in terms of interpretations. The semantics of dCSP is extended straightforwardly:

$$\mathcal{M} \models [P]F \quad \text{iff} \quad \mathcal{I} \models F \text{ holds for every interpretation } \mathcal{I} \in [\![P]\!]\mathcal{M}$$
$$\mathcal{M} \models \langle P\rangle F \quad \text{iff} \quad \mathcal{I} \models F \text{ holds for some interpretation } \mathcal{I} \in [\![P]\!]\mathcal{M}$$

6.2.1. Test Processes for Timing Properties

In the automata-theoretic verification approach [VW86], a property φ of an automaton $\mathcal{A}$ is verified by constructing an automaton for the negation of φ, $\mathcal{A}_{\neg\varphi}$, and showing that the parallel composition $\mathcal{A} \parallel \mathcal{A}_{\neg\varphi}$ is empty, i.e.,

$$\mathcal{A} \parallel \mathcal{A}_{\neg\varphi} \models \text{false}\,.$$

In [Mey05, MFHR08], this approach was carried over to DC test formulae and PEA. DC test formulae are a sub-class of DC, basically comprising conjunctions and disjunctions of negated DC counterexample traces, which are used to specify undesired behaviour of a PEA. In [Mey05] a translation of test formulae into PEA is given. To verify a test formula against a PEA model, a PEA with final locations, called *test automaton*, is constructed from the test formula. The parallel composition of this test automaton and the PEA model is computed. A model checker is then applied to check if the final locations are reachable. If this is not the case, the test formula is unsatisfiable.

The automata-theoretic approach can be applied to verify timed properties within dCSP. The corresponding proof rule is

$$\frac{[P \parallel P_{\neg F}]\mathit{false}}{[P]F}, \qquad \text{(box timed)}$$

where $P_{\neg F}$ is a process with the property that for all terminating interpretations $\mathcal{I} = \langle \ldots, \checkmark \rangle$ the equivalence

$$\mathcal{I} \in [\![P_{\neg F}]\!]\mathcal{M} \quad \text{iff} \quad \mathcal{I} \notin [\![F]\!]\mathcal{M} \qquad (6.25)$$

holds. Note that F must not constrain the $\checkmark$-event. In addition, the parallel composition $P \parallel P_{\neg F}$ is required to be fully synchronised, i.e., it synchronises on every event; by this the processes are also synchronised on the data, because the processes must agree on the data changes for every synchronisation.

For DC properties, we make use of the approach of [MFHR08]. We restrict ourselves to DC formulae without data. We consider a formula $F = \neg G$, where G is a DC test formula. The process $P_{\neg F}$ is then defined by

$$P_{\neg F} := \mathtt{Proc}_{\setminus \varnothing, \varnothing} \bullet \overline{G},$$

where $\overline{G}$ is the DC formula performing a $\checkmark$-event whenever G is recognised. That means that the interpretations of $\overline{G}$ are the interpretations of the corresponding automaton for G, constructed by the algorithm in [MFHR08], with the difference that an explicit $\checkmark$-event is fired whenever a final location of the automaton is entered.

Condition (6.25) is satisfied for $P_{\neg F}$: if a terminating interpretation $\mathcal{I} = \langle \ldots, \checkmark \rangle$ is in $[\![P_{\neg F}]\!]\mathcal{M}$, this implies $\mathcal{I} \models G$ and $\mathcal{I} \not\models \neg G = F$; and the same holds for the other direction.

$$\frac{\vdash [P]F_1 \wedge [Q]F_2}{\vdash [P \mathbin{;} Q]F_1 \frown F_2} \qquad \text{(dc sequence)}$$

$$\frac{\vdash [P]F_1 \wedge [Q]F_2}{\vdash [P \mathbin{;} Q](F_1 \frown \bigwedge_{a \in Events} \not\updownarrow a \frown F_2)} \qquad \text{(dc sequence noev)}$$

$$\frac{\vdash \langle\!\langle P \rangle\!\rangle\, G}{\vdash \langle\!\langle P \rangle\!\rangle\, F,} \qquad G \Rightarrow F \qquad \text{(dc imply right)}$$

$$\frac{\langle\!\langle P \rangle\!\rangle\, G \vdash}{\langle\!\langle P \rangle\!\rangle\, F \vdash,} \qquad F \Rightarrow G \qquad \text{(dc imply left)}$$

$$\frac{}{[\mathtt{Proc}_{\setminus A, V} \bullet F]F} \qquad F \text{ respects } A \text{ and } V \qquad \text{(dc uproc)}$$

$$\frac{[P]F_1 \wedge [Q]F_2}{[P \parallel Q]F_1 \wedge F_2} \qquad P, Q \text{ without shared variables} \qquad \text{(dc parallel)}$$

$$\frac{\psi_1 \wedge [a \bullet \varphi]\psi_2}{[a \bullet \varphi](\lceil \psi_1 \rceil \frown \updownarrow a \frown \lceil \psi_2 \rceil)} \qquad \text{(dc step)}$$

$$\frac{}{[a \bullet \varphi](\boxminus\, b)} \qquad a \neq b \qquad \text{(dc noevent)}$$

Figure 6.2.: Sequent-style proof rules for timing properties

The property $[P \parallel P_{\neg F}]$*false* can be verified with the translation-based approach for parallel components of Sect. 6.1.2 by applying rule (parallel uproc). To this end, the *ProcFree* process is to be constructed according to the translation of Sect. 6.1.2 with the exception that $\overline{G}$ and *ProcFree* are translated into a test automaton, which has final locations. With this modification, the resulting *ProcFree* process terminates exactly for the interpretations that violate F. Thus, by verifying $[P \parallel P_{\neg F}]$*false*, it is actually shown that F holds for all interpretations.

For DC formulae with data, we cannot directly carry over the test automata approach of [MFHR08], because it is directed to data-synchronous PEA, while the

parallel composition in eCSP does usually not synchronise on data (cf. Sect. 3.2.2). Hence, in the construction of the corresponding test process $P_{\neg F}$, one has to take care that all data changes are actually synchronised, even for transitions that are not event-triggered. The case $\langle P \rangle F$ can be solved by verifying the dual $\neg[P]\neg F$, which is possible with the proposed technique above if F is a DC counterexample trace. Then, the negation $\neg F$ is still a DC test formula, which is necessary for the translation into PEA.

6.2.2. Extended Calculus for Timing Properties

We sketch the idea how proof rules over timing properties are integrated into our proof calculus using DC as example. Figure 6.2 lists some exemplary proof rules for solving DC properties over eCSP processes.

One of the central concepts of sequent-style calculi is that operators of the logic are step-wise replaced by simpler constructs. The basic operator of DC is the *chop* operator $\frown$, which is used to chop an interval into two sub-intervals. The corresponding operation on the level of eCSP is the sequential composition. With rule (dc sequence) the relationship between the chop operator and sequential composition is expressed: we can reduce the chop operator if two sequential sub-processes can be found that fit to the sub-intervals of the chop. A variant is rule (dc sequence noev), which is necessary to handle formulae that forbid the occurrence of events.

The reduction of rule (dc sequence) seems to be quite natural in the sense of sequent-style reasoning. However, often properties need to be proven that are not in the shape of the conclusion of the rule. For instance, a typical property to be proven is

$$\vdash [\texttt{Proc} \bullet \ell < c_1 \mathbin{;} \texttt{Proc} \bullet \ell < c_2](\ell < c_1 + c_2),$$

where $\ell < c_1 + c_2$ is not of the desired shape. Thus, we also need rules embedding standard DC proof rules like the rules that can be found in [ZHR91, ZH04, OD08]. To this end, the rules (dc imply right) and (dc imply left) can be used to derive pure DC properties with standard rules. As example consider the following proof tree:

$$\cfrac{\cfrac{\cfrac{\cfrac{\text{(dc uproc)}}{\vdash [\texttt{Proc} \bullet \ell < c_1](\ell < c_1)} \qquad \cfrac{\text{(dc uproc)}}{\vdash [\texttt{Proc} \bullet \ell < c_2](\ell < c_2)}}{\vdash [\texttt{Proc} \bullet \ell < c_1](\ell < c_1) \wedge [\texttt{Proc} \bullet \ell < c_2](\ell < c_2)}\text{(and right)}}{\vdash [\texttt{Proc} \bullet \ell < c_1 \mathbin{;} \texttt{Proc} \bullet \ell < c_2](\ell < c_1 \frown \ell < c_2)}\text{(dc sequence)}}{\vdash [\texttt{Proc} \bullet \ell < c_1 \mathbin{;} \texttt{Proc} \bullet \ell < c_2](\ell < c_1 + c_2)}\text{(dc imply right)}$$

In the first proof step (starting from the bottom) rule (dc imply right) is applied to replace $\ell < c_1 + c_2$ by $\ell < c_1 \frown \ell < c_2$, which is a consequence of the *Dur-Chop* rule of [OD08]. The leafs of the proof tree are closed by application of rule (dc uproc), stating that for every unknown process $\texttt{Proc}_{\setminus A,V} \bullet F$ the formula F is valid, as long as F does not perform an event from A and does not change symbols in V.

Further important rules reduce parallel compositions and symbolically execute constrained events. For the reduction of the parallel composition, it is desired to replace the parallel composition operator with a conjunction. However, the same issues exist as for the case without timing properties, which have been discussed in Sect. 6.1: for parallel processes with shared variables, one has to prove that no unwanted interferences between the processes occur. This can be done with appropriate Rely-Guarantee rules or by manually excluding interferences between the processes. See, e.g., [XM98] for an A-G approach wrt. DC formulae. Rule (dc parallel) in Fig. 6.2 considers the simple case that both processes have no shared variables and are fully synchronised on all events.

The last two rules (dc step) and (dc noevent) represent the symbolical execution of events for DC properties. The rule (dc step) replaces the timing property by standard dCSP formulae that can be solved with the calculus of Sect. 4.3. The DC formula in the conclusion reflects the intuition behind the symbolic execution of a: before occurrence of event a, a formula ψ_1 holds, which is valid in the current context of the rule, and after execution of a, a formula ψ_2 is valid. Rule (dc noevent) is used to derive a DC formula expressing that an event b does not occur.

Discussion

The proof rules of Fig. 6.2 exemplarily show for the DC how timing properties over eCSP can be proven within the sequent calculus. One of the main reasons for introducing sequent-style proof rules for dCSP has been that the sequent rules allow us to step-wise reduce operators symbolically. By this means, suchlike sequent calculi are suited for a systematic, semi-automatic application of the proof rules, because in most cases, it is clear which syntactical reduction is to be performed in the next step. That gives rise to successful implementations of sequent calculi as existing tools demonstrate [HBB+05, BHS07, PQ08]. However, when extending the calculus to DC properties these advantages of sequent-style reasoning do no apply anymore. Applications of rules like rule (dc sequence) are often ambiguous, because it has to be guessed, at which position the DC formula has to be chopped or which sequential operator has to be resolved. This holds particularly for longer sequences or formulae, e.g., $[P_1 \fatsemi P_2 \fatsemi P_3](F_1 \frown F_2 \frown F_3 \frown F4)$. The situation becomes even more complex for the diamond operator: for $\langle P_1 \fatsemi P_2\rangle(F_1 \frown F_2)$ one has to find a constraint φ that is valid after F_1 and implies $\langle P_2\rangle F_2$.

Moreover, as the DC formulae in the conclusion are often of a specific shape, the rules (dc imply right)and (dc imply left) have to be applied often in order to transform the formulae into the desired form, which usually cannot be done as systematic as in remaining sequent calculus. For instance, even in the small example proof tree above, backward application of rule (dc imply right) does not lead to a decomposition into a less complex formula. It is not clear how to apply this rule in general.

Therefore, solving DC properties in the sequent calculus might not be the right choice. Since safety properties are usually expressed in terms of untimed formulae,

we do not examine in more detail how timing properties can be integrated into our calculus.

6.3. Examining Completeness

We now examine in which situations and under which circumstances the proposed VA approach can be considered complete. There are several levels on which one can look at completeness questions:

Completeness of VA language. It is desirable that all relevant specifications can be expressed as VA. This depends on what kind of specifications are considered to be relevant. Since the VA approach presented here is designed to complement combined specifications, we are particularly interested in the completeness of our VA approach with respect to CSP-OZ-DC. That is, we examine whether every CSP-OZ-DC specification can be equivalently expressed as VA.

Completeness of local assumptions. More important than the completeness of the VA language is the completeness of the local assumptions. The former answers the question if for all possible specifications an equivalent VA can be given, but this does reveal nothing about the quality of the architecture. So, we examine the completeness of the local assumptions, i.e., the possibility to find local assumptions for every abstract behavioural protocol with unknowns and a corresponding concrete structural refinement. By this, we show that our approach is strong enough to formulate every reasonable decomposition in a VA. This notion of completeness depends on the chosen temporal logic for the local assumptions. Thus, we exemplarily choose DC as logic and discuss the completeness of the local assumptions for DC formulae.

Completeness of the proof calculus. With a complete proof calculus, we can prove every statement that is valid for a specific architecture within our calculus. We have shortly argued on the incompleteness and the possible relative completeness of the calculus in Thm. 4.4.2.

Completeness of instantiation. Completeness of instantiation means: if a concrete model is an instantiation of an VA, then we can actually prove it. As elucidated in Sect. 5.1, we usually apply an efficient but incomplete syntactic rule to establish the structural refinement between a concrete model and a VA. We have discussed in Sect. 5.6 how to achieve a complete refinement rule that can be used to prove every refinement relation.

To show that a concrete specification is an instance of a VA, we additionally need to verify the local assumptions. This solely depends on the decidability of the verification problem for the logic of the assumptions and the concrete model. The model checking problem for DC and CSP-OZ-DC is for instance

undecidable [Hoe06]. When using timed automata and timed CTL as logic, it is decidable [ACD93].

Particularly, the second topic, the completeness of local assumptions, gives rise to the compositionality of the VA approach. If a behavioural protocol with unknowns and a corresponding concrete model is given, then it is possible to find a set of local assumptions for the refining components. In general, these assumptions can be arbitrary complex, because in the worst case they need to reflect the behaviour of the entire refining component. Hence, to obtain assumptions that are useful in practice one needs to find assumptions that are minimal (or at least small enough) with respect to a desired global safety property. But as we assume VAs to be user-given, the automatic assumption generation is out of the scope of this work, and we take this completeness result as a confirmation that it is indeed possible to give appropriate local assumptions for every unknown component. For results on automated generation of assumptions for components in networks of timed systems see, e.g., [FPS08, FPS10].

In the following, we discuss those completeness issues that are not already treated in Chap. 4 or Chap. 5, namely the completeness of the VA language and the completeness of the local assumptions.

6.3.1. Completeness of VA Language

Every CSP-OZ-DC class can equivalently be expressed as VA with DC as logic for assumptions. The idea is as follows. The CSP part of the CSP-OZ-DC specification is exactly the CSP part of the architecture without unknown processes. The formulae of the constrained events of the VA correspond directly to the operation schemata of the CSP-OZ-DC class. If the CSP-OZ-DC class contains DC formulae, then we add one parallel unknown process constrained by the conjunction of all DC formulae of the CSP-OZ-DC specification.

More formally: consider a CSP-OZ-DC specification *cod* as defined in Fig. 6.3. Here, φ, ψ, δ are OZ constraints as allowed in CSP-OZ-DC specifications. The equivalent VA is then given as the eCSP process *System* defined by

$$\begin{aligned} \mathit{System} \overset{c}{=} \mathit{init} \bullet \eta \to (&P(a_1, \ldots, a_n)[(a_1 \bullet \gamma_1)/a_1, \ldots, (a_n \bullet \gamma_n)/a_n] \\ &\underset{\{a_1,\ldots,a_n\}}{\parallel} \ \mathtt{Proc}^{\infty}_{\setminus \varnothing, \varnothing} \bullet DC_1 \wedge \ldots \wedge DC_r), \end{aligned}$$

where $Q[new_1/old_1, \ldots, new_n/old_n]$ denotes the replacement of all old_i by new_i in the process Q, and *init* is a fresh event not occurring elsewhere. All γ_i and η are formulae of the signature $\Sigma = (Sort, SysVar, Const, Var)$ with

$$\begin{aligned} Sort &= \{T_1, \ldots, T_m\} \cup \bigcup_{i \in 1..n} \{T^i_1, \ldots, T^i_{k_i}\} \cup Sort_{Const} \\ SysVar &= \{v_1, \ldots, v_m\} \\ Var &= \bigcup_{i \in 1..n} (\{p^i_1, \ldots, p^i_{k_i}\} \cup Var(\delta^i)) \cup Var(\psi) \end{aligned}$$

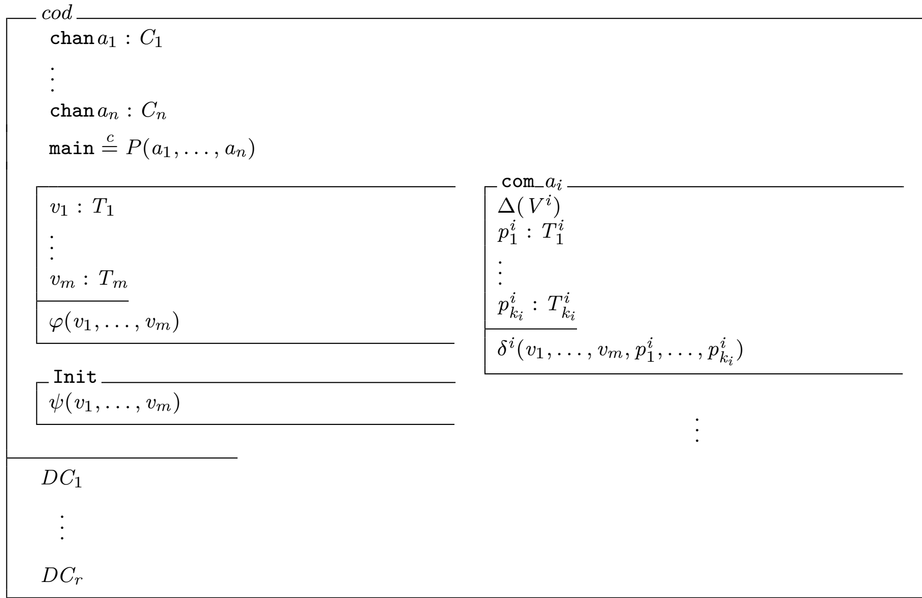


Figure 6.3.: Generic CSP-OZ-DC specification *cod*

The set *Const* contains global constants with sorts in $Sort_{Const}$ which are defined in the context of a CSP-OZ-DC specification, including all function symbols used in the constraints of the class *cod* (like, e.g., arithmetical symbols). The γ_i and η are defined by

$$\gamma_i := \delta^i(v_1, \ldots, v_m, p^i_1, \ldots, p^i_{k_i}) \wedge \bigwedge_{v \notin V^i} v' = v$$

$$\eta := \psi(v_1, \ldots, v_n) \wedge \bigwedge_{i \in 1..m} v'_i = v_i$$

Note that the specification *cod* and the *System* process produce the same interpretations except for the initial model. Due to the eCSP semantics, the *System* process may start in an arbitrary model, but the *init* event can only be executed for models satisfying the `Init` constraint of *cod*. Thus, the interpretations coincide except for trivial, immediately blocking interpretations of the shape $\langle \mathcal{M} \rangle$.

If we have a system of two CSP-OZ-DC classes cod_1, cod_2 communicating over common channels from a set C, then we compute the CSP representations $System(cod_1)$ and $System(cod_2)$ for each of the processes according to the construction above. If cod_1 and cod_2 share variable names, the variables in one of the processes have to be renamed into fresh variable names in order to avoid shared variable access, which is not possible between CSP-OZ-DC classes. The signature consists of the union of the signatures of the $System(cod_i)$, and the entire CSP process is the parallel composition

of both processes:

$$System := System(cod_1) \underset{C}{\parallel} System(cod_2).$$

With this construction, we can rewrite every CSP-OZ-DC specification into a CSP process with data constraints and unknown processes. In this sense, we consider our language to define VAs as complete. This is a weak form of completeness due to its focus on CSP-OZ-DC. One could also show in general that every set of timed traces can be modelled with eCSP and an appropriate timed logic. With a sufficient expressive timed logic this question can be trivially answered by a VA process consisting of one unknown component that is appropriately constrained by an assumption.

But as mentioned in the beginning of this section, we are not interested in modelling arbitrary specifications in our CSP dialect; instead eCSP processes are used to formulate architectures for the decomposition of large systems. Hence, we are satisfied with this weak notion of completeness that shows that the extended CSP dialect eCSP is strong enough to be useful. We examine in the next section to what extent eCSP can be used to formulate every reasonable decomposition of a system.

6.3.2. Completeness of Local Assumptions

We assume to have an abstract protocol given as CSP process with unknown processes but without temporal assumptions on the unknown processes. Further, we consider a concrete structural refinement of this protocol. We investigate the question whether it is always possible to find appropriate local assumptions on the unknown processes such that the concrete model is an instantiation of the protocol. By this, it becomes clear that whenever a decomposition by a protocol is given, and the protocol itself is strong enough to not violate a desired property, then it is possible to find suited local assumptions.

The short answer to that question is, that this will not work in general, but we will figure out that it is possible with small modifications to the architecture. These modifications are necessary, because the only way to define timing properties for eCSP processes is to constrain unknown processes with timed formulae. Thus, only in the case that the unknown processes of the VA fit directly to the timing properties of the concrete model, it is possible the find appropriate timing properties on the VA level.

Example 6.3.1. To give an example for this issue, we consider the CSP process for an abstract protocol

$$System \overset{c}{=} a \to \mathtt{Proc}_{\setminus\{a,b\},\varnothing} \fatsemi b \to System.$$

If we have a concrete specification with a control structure given as a CSP process

$$\mathtt{main} \overset{c}{=} a \to c \to \mathtt{Skip} \fatsemi b \to \mathtt{main},$$

which is a structural refinement of *System*, and an additional timing constraint expressed as DC formula

$$F := \Box\neg(\updownarrow a ^\frown \boxminus b \wedge \ell > 5),$$

stating that every a is followed by a b within 5 time units, then we cannot find a constraint for the unknown process in *System* such that timing constraint F holds, because the events constrained by F are not in the scope of the only unknown process.

The way out of this is a small modification to *System* to capture all possible timing constraints with an additional unknown process. If the protocol is given by an extended process of the shape

$$\mathtt{Proc}_{\backslash\varnothing,\varnothing} \underset{a,b}{\parallel} System,$$

then we can directly shift the timing constraints of `main` to the VA level such that `main` is an instance of an appropriate abstract protocol. That is, `main` is an instance for the following VA process with constraints:

$$System_{ext} \stackrel{c}{=} \mathtt{Proc}_{\backslash\varnothing,\varnothing} \bullet F \underset{a,b}{\parallel} System$$

$$System \stackrel{c}{=} a \rightarrow \mathtt{Proc}_{\backslash\{a,b\},\varnothing} \fatsemi b \rightarrow System.$$

Generally speaking, if the concrete specification contains timing constraints, then the architecture must cover each timing property with a specific unknown process.

We now show for CSP-OZ-DC how the assumptions can be constructed for a structural refinement of a given VA process without assumptions.

Assumption completeness for specifications without DC part. Let *co* be a CSP-OZ-DC specification without DC part and *Prtcl* an eCSP process with unknowns $X_i \stackrel{c}{=} \mathtt{Proc}^{(\infty)}_{\backslash A_i,V_i}$ for $i \in 1..n$ (without constraints over the unknowns) such that *co* refines *Prtcl*. That is, *co* contains sub-processes $Y_1, \dots, Y_n$ implementing the unknown processes $X_1, \dots, X_n$. We construct an equivalent DC formula F_i for the reduction of *co* to the sub-process Y_i. The reduction $co_{|Y_i}$ has been defined in Def. 5.5.1 and reflects exactly the sub-specification that refines an unknown process X_i and for which the local assumptions must be valid.

To generate a DC formula with the same interpretations as a CSP-OZ-DC specification without DC part, the specification is firstly translated into two PEA: one PEA represents the control flow, and the second describes all data changes (Sect. 2.3.2). These automata can be translated into DC formula analogously to Anders P. Ravns translation from control automata into DC implementables [Rav94], which can be represented as DC counterexample traces [Hoe06].

For the control flow PEA, a program counter *pc* is introduced to identify the locations of the PEA. For a location l_1 and an event a with outgoing transitions $l_1 \xrightarrow{a} l_2$

and $l_1 \xrightarrow{a} l_3$ (and there are no other outgoing a-transitions), an equivalent DC formula is defined by

$$\neg\Diamond(\lceil pc = l_1 \rceil ^\frown \updownarrow a ^\frown \lceil \neg pc = l_2 \wedge \neg pc = l_3 \rceil).$$

Transitions that are not event-triggered are specified by

$$\neg\Diamond(\lceil pc = l_1 \rceil ^\frown \lceil \neg pc = l_1 \wedge \neg pc = l_2 \wedge \neg pc = l_3 \rceil).$$

If the event a cannot occur in the location l_1, the following DC formula is used:

$$\neg\Diamond(\lceil pc = l_1 \rceil ^\frown \updownarrow a ^\frown \mathit{true}).$$

Transitions in the second PEA are event-triggered and describe state changes corresponding to operation schemas. For an event a describing a data change φ over unprimed variables $\overline{x}$ and primed variables $\overline{x}'$, the corresponding the DC formula is defined by

$$\forall\, y \bullet \neg\Diamond(\lceil \varphi^{y}_{x'} \rceil ^\frown \updownarrow a ^\frown \lceil x \neq y \rceil),$$

where in $\varphi^{y}_{x'}$ variables x' are replaced by fresh variables y. The conjunction of all DC formulae constructed in this way equivalently describes the CSP-OZ-DC specification.

Note that due to the quantification over y the latter DC formula cannot be verified in negated form with the approach from [MFHR08]. A consequence is that suchlike assumptions *can* be verified against concrete CSP-OZ-DC models when checking the instantiation of a VA, but the assumption cannot be verified with the oracle rules of Sect. 4.3.5 when proving the correctness of a VA with our sequent calculus. The reason is that in this case, we need to show a certain property while postulating that the assumptions hold, i.e., the assumptions occur in negated form in this verification step. This is not a restriction of the VA approach but of the DC verification technique we use. Nevertheless, instead of using assumptions as generated above, in practice one will use more abstract assumptions that do not exactly reflect the concrete process but instead a class of processes.

The extension of the process *Prtcl* to a VA process *va* with additional assumptions on the unknowns, is defined by

$$va \stackrel{c}{=} \mathit{Prtcl}[\overline{X}_1/X_1, \ldots, \overline{X}_n/X_n]$$
$$\overline{X}_i \stackrel{c}{=} \mathtt{Proc}^{(\infty)}_{\setminus A_i, V_i} \bullet F_i,$$

where F_i is the DC formulae for $co_{|Y_i}$ as defined by the construction above. Then, *co* refines *va*, because it is a structural refinement of *va* by precondition, and all local assumptions are satisfied by construction of the F_i.

Assumption completeness for CSP-OZ-DC specifications. If a CSP-OZ-DC specification *cod* with DC formulae $G_1, \ldots, G_m$ is given, we use the same construction

as in Example 6.3.1. That is, we demand that the VA process consists of a parallel composition

$$Prtcl_0 \overset{c}{=} Ptrcl \parallel \mathtt{Proc}^{(\infty)}_{\backslash\varnothing,\varnothing}.$$

The additional unknown process serves as a carrier process for all DC formulae from *cod*. The remaining construction is identical to the case without time. Let *co* be the specification that is equal to *cod* except that the DC formulae are removed. Then the VA for *cod* is defined by

$$va \overset{c}{=} Prtcl[\overline{X}_1/X_1, \ldots, \overline{X}_n/X_n] \parallel \mathtt{Proc}^{(\infty)}_{\backslash\varnothing,\varnothing} \bullet G_1 \wedge \ldots \wedge G_m$$
$$\overline{X}_i \overset{c}{=} \mathtt{Proc}^{(\infty)}_{\backslash A_i, V_i} \bullet F_i,$$

where the parallel composition synchronises on all common events, and the F_i are DC formulae representing $co_{|Y_i}$ according to the construction above. Due to this construction, the process *va* is an architecture for *cod*.

Thus, we have demonstrated in this section to what extent the VA language eCSP and the local assumptions are complete, i.e., we have examined (1) whether all relevant specifications can be formulated as VA and (2) whether assumptions can be found for all given concrete models that are structural refinements of VA processes. As these questions depend on the logic used for the assumptions and on the language for the concrete model, we have exemplarily showed that all CSP-OZ-DC specifications can be transformed into an eCSP process and that DC assumptions for all CSP-OZ-DC models can be found if we modify the VA process by adding an unknown component with appropriate DC formulae. Otherwise, it is generally not possible to find sufficient strong assumptions on the unknown components such that all desired properties that hold for the concrete model can be shown for the VA, because timing properties in VAs can only be defined over unknown processes.

The construction that we give for finding DC assumptions for a CSP-OZ-DC specification yields assumptions usually consisting of a large number of sub-formulae, because they describe all possible transitions of the corresponding CSP-OZ-DC specification. This suffices for our purpose to demonstrate that the VA language is general enough to describe architectures for a large class of concrete systems. Since in this work we require the assumptions to be given by a user, we do not further investigate how to get *good* assumptions and instead refer to [FPS08, FPS10].

Further completeness questions, the completeness of the proof calculus and of the instantiation approach, have been discussed in Chap. 4 and Chap. 5.

6.4. Complementary Decomposition Techniques

In this section, we shortly review two decomposition techniques that can be used as complementary techniques in combination with our approach: *slicing* of formal specifications and the *communication-closed layers* principle.

6.4.1. Slicing Formal Specifications

Slicing for CSP-OZ-DC specifications has been developed by Ingo Brückner and Heike Wehrheim [Brü04, BW05, BMW06, Brü07, Brü08a, Brü08b]. The slicing decomposition technique is based on a syntactic dependency analysis that has originally been developed in the context static program analysis and later been extended with respect to formal verification techniques.

The basic idea of slicing specifications is that dependency graphs are computed considering only the syntactic structure of a specification and also exploiting additional knowledge on the system structure that often gets lost when translating the specification into its semantical representation. In a first step, a directed control dependency graph is calculated, reflecting the control flow of the specification. Afterwards, the control dependence graph is enriched with data dependencies, and when considering timed CSP-OZ-DC specifications, also with timing dependencies computed from the DC part. Given a so-called *slicing criterion*, which is a property that is to be verified, e.g., a DC formula, all nodes in the dependence graph are selected that directly influence the slicing criterion. Starting with these nodes, the set of all nodes relevant for the slicing criterion can be detected by computing the backward reachability closure for these nodes. Brückner proved in [Brü08b] that the correctness of the slicing criterion is preserved when removing all parts of the specification that are not in this closure, because they have no influence on the property.

Slicing is a very efficient technique, because it operates directly on the syntactic structure of a specification, i.e., no semantics or parallel composition has to be computed. In [Brü08b], the advantages of slicing are demonstrated on several examples. Slicing is particularly successful on real-world models, where often large parts of a specification have no influence on its safety. For instance such specifications contain pure output or formatting functions that can be ignored when verifying the system.

On the contrary, when verifying examples that are designed with regard to a given safety property, slicing has a lesser effect, as in this case usually all parts of the specification affect the desired property. In general, slicing cannot be used for further decomposition of specifications, in which each part contributes to the desired property in an arbitrary marginal manner. In the examples of Chap. 8 slicing cannot be used to reduce the specification significantly when checking the desired safety property directly.

However, when verifying a global property with the VA approach, the property is not checked against the entire specification in a single step. Instead the local assumptions of the VA are verified against the sub-specifications that refine the unknown parts of the VA. In these sub-specifications parts of the control flow are often not reachable or variables are not accessed, because they are only modified in other subprocesses. Additionally, local assumptions usually comprise other variables than the global property and are influenced by smaller parts of the system. Thus, when verifying the local assumptions, slicing can be used to automatically remove parts that are independent of the current examined sub-process. The slicing approach therefore

complements the VA approach by reducing the specification for proof tasks even if slicing for the global property has no effect. The verification results for the case study presented in Sect. 8.2 demonstrates the advantages of using slicing in combination with the VA approach.

6.4.2. Layered Composition for Timed Protocols

We have discussed in Sect. 4.6.1 that the VA approach works out particularly well on sequential protocol structures. Even though we showed in Sect. 6.1 how to cope with parallel compositions, VAs are desired that are as sequential as possible. Thus, it is an interesting question how a parallel protocol structure can be transformed into a sequential structure. An approach that examines this question is the *communication-closed layering* (CCL) principle [EF82, dRdBH+01], which is extended to timed systems in [Zwi91].

In [OS10], decompositional layering for timed automata with data, similar to PEA, is investigated. The basic idea is to transform parallel automata into sequential automata by using laws for a dedicated layered composition operator $\mathcal{A}_1 \bullet \mathcal{A}_2$ such that the result is reachability-equivalent to the origin automata. This operator expresses that an operation of $\mathcal{A}_2$ can only be executed after all dependent operations of $\mathcal{A}_1$ have been executed, where dependency means mutual access to shared variables (read/write or write/write).

For the layered composition operator, the following CCL law is defined:

$$(\mathcal{A}_1 \bullet \mathcal{A}_2) \parallel (\mathcal{B}_1 \bullet \mathcal{B}_2) \equiv (\mathcal{A}_1 \parallel \mathcal{B}_1) \bullet (\mathcal{A}_2 \parallel \mathcal{B}_2),$$

in which $\equiv$ denotes reachability equivalence. A further side condition is that either (1) every operation in $\mathcal{A}_1$ precedes (wrt. time) dependent operations in $\mathcal{B}_2$, the same holds for $\mathcal{B}_1$ and $\mathcal{A}_2$, and all automata are acyclic, or (2) the operations of $\mathcal{A}_1$ and $\mathcal{B}_2$ are independent as well as the operations of $\mathcal{B}_1$ and $\mathcal{A}_2$.

Using the CCL law, parallel compositions can be transformed into a sequential composition:

$$\begin{aligned} &(\mathcal{A}_1 \fatsemi \mathcal{A}_2) \parallel (\mathcal{B}_1 \fatsemi \mathcal{B}_2) \\ \equiv\ &(\mathcal{A}_1 \bullet \mathcal{A}_2) \parallel (\mathcal{B}_1 \bullet \mathcal{B}_2) \\ \equiv\ &(\mathcal{A}_1 \parallel \mathcal{B}_1) \bullet (\mathcal{A}_2 \parallel \mathcal{B}_2) \\ \equiv\ &(\mathcal{A}_1 \parallel \mathcal{B}_1) \fatsemi (\mathcal{A}_2 \parallel \mathcal{B}_2), \end{aligned}$$

where $\equiv$ is a suited equivalence preserving the desired properties to be shown (in [OS10] partial-order equivalence is used, which preserves LTL properties). For the second equivalence the CCL law is applied. By this transformation, the costly parallel composition operator is applied to smaller automata, which leads to a reduction of the state space that is to be analysed. See [OS10] for details on the approach.

The layered composition Principle is a promising technique that helps to find good VAs for concrete systems or to restructure VAs into a layered protocol that can be

applied more efficiently. To this end, protocol processes are transformed into the shape of $(\mathcal{A}_1 \parallel \mathcal{B}_1) \fatsemi (\mathcal{A}_2 \parallel \mathcal{B}_2)$ by applying the CCL transformation. The parallel compositions $(\mathcal{A}_i \parallel \mathcal{B}_i)$ are natural candidates for protocol phases in a VA, modelled as constrained unknown processes. By this means, the sequential parts of the VA are handled at the eCSP level, whereas the layers containing the parallel parts are verified when instantiating the VA process by checking the local assumptions. It is future work to investigate the details on using the layered composition principle to derive suited VA processes.

7 Implementation and Tools

> Oh, but it is true. Things need not have happened to be true. Tales and dreams are the shadow-truths that will endure when mere facts are dust and ashes, and forgot.
>
> *(Dream, in* Dream Country*, Neil Gaiman)*

7.1. Syspect . . . 174
- 7.1.1. UML Profile for Real-Time Systems . . . 174
- 7.1.2. Tool Structure . . . 178
- 7.1.3. Syspect Plug-Ins . . . 179

7.2. Verification with Syspect . . . 181
- 7.2.1. Transition Constraint Systems . . . 181
- 7.2.2. Verification of Syspect Specifications . . . 182
- 7.2.3. Slicing CSP-OZ-DC Specifications in Syspect . . . 185
- 7.2.4. Further Verification Plug-Ins . . . 185

7.3. Syspect Verification Architecture Plug-In . . . 185
- 7.3.1. Modelling of VAs . . . 185
- 7.3.2. CSP-OZ-DC Representation of a VA . . . 186
- 7.3.3. Verification of VAs . . . 187

7.4. Discussion . . . 189

As this work is focused on automated verification of complex real-time systems, tool support is essential for the successful application of the VA approach to large systems. Even though there is still no tool support for every aspect introduced in this thesis,

a tool chain is introduced in this chapter that allows for automatic verification of real-time systems and that supports Verification Architectures.

The following section introduces the tool *Syspect*, which has been developed in the context of this work and which has been used to model and verify the examples of Chap. 8. Afterwards, a Verification Architecture plug-in for Syspect is described.

7.1. Syspect

Syspect [FLOQ11] is a graphical UML front-end for CSP-OZ-DC that has initially been developed as an one-year collaborative student project at the University of Oldenburg [Sys06]. It has been further improved in Oldenburg as part of the AVACS project and in the research group "Specification and Modelling of Software Systems" of Heike Wehrheim in Paderborn. Syspect is an open source project distributed under the GNU Public License (GPL) and its sources as well as pre-compiled binaries for Windows, Mac OS, and Linux are available at

`http://syspect.informatik.uni-oldenburg.de/`.

7.1.1. UML Profile for Real-Time Systems

The *Unified Modelling Language* (UML) [RJB99, OMG09] is diagram-based language that is widely used by engineers to describe complex systems. However, these diagrams generally have no formal semantics and, thus, are not suited for formal analysis and verification. To overcome this drawback, [MORW08] introduced a dedicated *UML profile* with a semantics in terms of CSP-OZ-DC. With a UML profile the UML can be adjusted to a specific application area by defining so-called *stereotypes* indicating a specific role of a UML element in the desired application context.

In [Sys06, FLOQ11], the stereotypes *capsule*, *data*, and *interface* for classes of reactive systems are defined:

- a UML class that is distinguished by the stereotype *capsule* has a semantics in terms of a CSP-OZ-DC class, particularly, capsules describe the control flow, state changes, and the timing behaviour of an entity
- a *data* class defines complex data types, and it has a semantics in terms of an OZ class, i.e., it defines operations on data, but in contrast to capsules it does not constrain the control structure or the timing behaviour
- classes that are distinguished by the stereotype *interface* describe the operations that are provided and shared between several capsules and that are used for synchronisation.

This UML profile is implemented in Syspect. It incorporates three types of diagrams: *class diagrams* to model the static structure of the system, *protocol state*

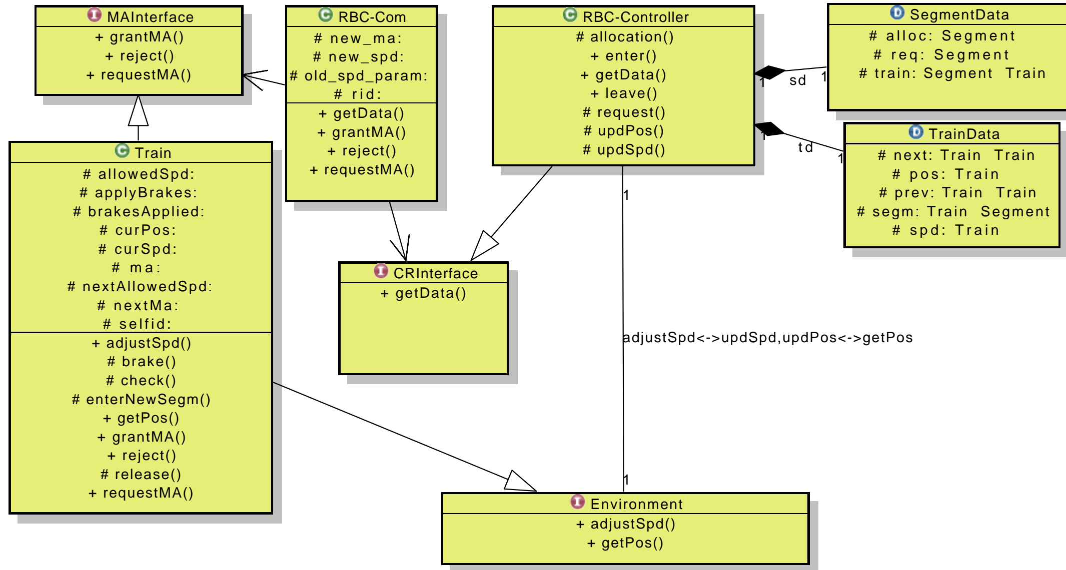


Figure 7.1.: An exemplary class diagram

machines, which are associated to capsules and which are used to model the control flow of capsules, and *component diagrams* to define communication structures between capsules and instances of capsules in a concrete system.

Figure 7.1 exemplarily pictures a class diagram for a case study from [FIJSS10a]. It is modelled with Syspect using the three stereotypes: Train, RBC-Com, and RBC-Controller are capsules (marked with an icon with a C); MAInterface, CRInterface, and Environment are interfaces (marked with an I icon), and SegmentData and TrainData are data classes. The connections in the class diagram are the usual UML connections, representing inheritance, composition, and dependencies.

The Train capsule has an associated state machine to define its control structure. It is pictured in Fig. 7.2 and consists of locations and transitions that are labelled with operation names, which have to be declared in the corresponding capsule. An operation may only be executed if the current location of the state machine has an outgoing transition labelled with the operation name (with the exception that an event that does not occur in the alphabet of the state machine can always occur, which often simplifies modelling). The filled black circles are start locations of the state machine and the dashed lines separate three hierarchical sub-machines that are executed concurrently.

The last diagram is the component diagram, pictured in Fig. 7.3, defining the instances of the capsules and their connections constituting the entire system. The example diagram connects an RBC-Com object via the MAInterface with two Train components. Particularly, the trains do not have access to the operations of RBC-Controller, because they are not connected in the diagram. The System component is

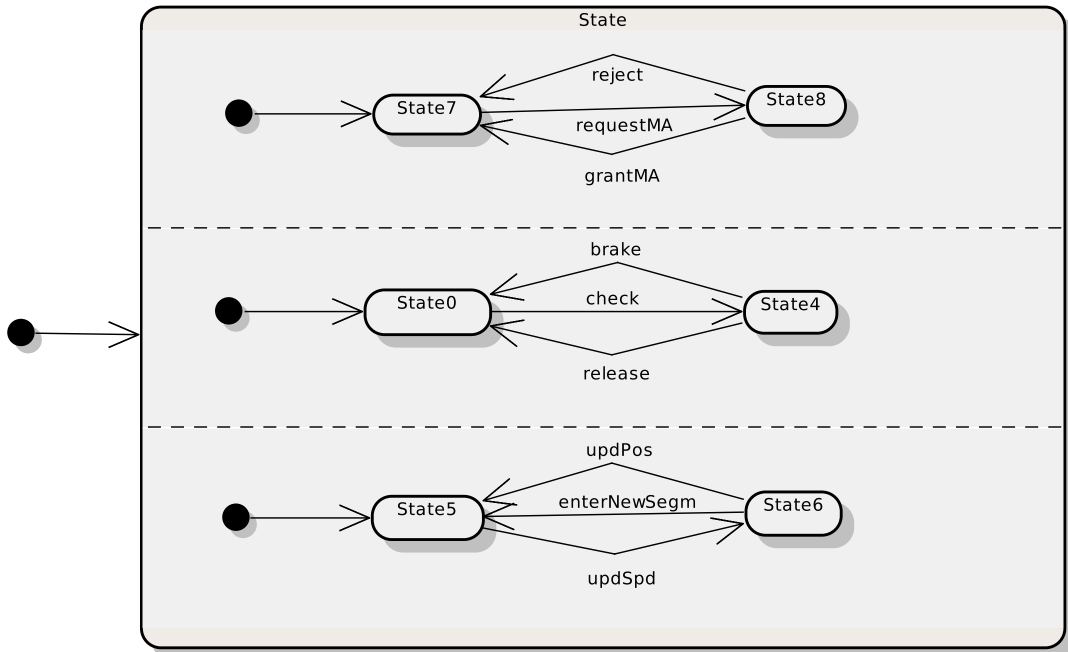


Figure 7.2.: State machine for Train

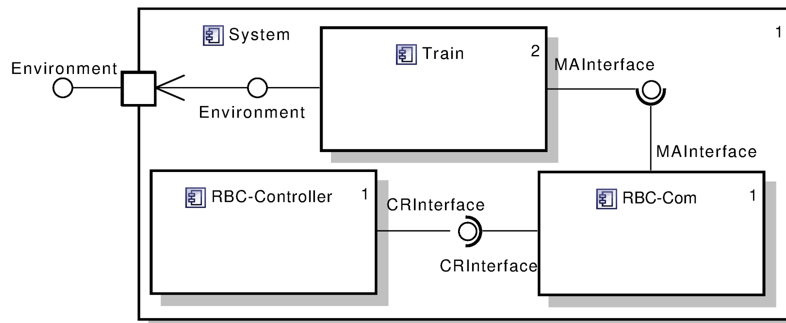


Figure 7.3.: Exemplary component diagram

Table 7.1.: Tags associated with operations (in capsules as well as data classes)

Tag	Purpose	Type
in	input parameter	Variable declaration
out	output parameter	Variable declaration
simple	general parameter	Variable declaration
changes	changed variables	list of variable names
enable	constraint that enables an operation	a Z constraint without primed variables
effect	constraint describing the effect of an operation	a Z constraint with primed variables

Table 7.2.: Tags associated with classes

Tag	Stereotype	Purpose	Type
init	*capsule*, *data*	initial constraint	Z predicate
invariant	*capsule*, *data*	invariant constraint	Z predicate
dc counterexample	*capsule*	timing behaviour	DC counterexample trace

a special component representing the entire specified system. In addition, the diagram specifies that any access to methods from the Environment interface is delegated to the train objects.

An important feature of Syspect is that class diagrams provide a particular view on the system model that does not need to reflect the entire model. Instead Syspect distinguished between model elements and views to model elements. By this, it is possible to cope with large system models by providing many diagrams, all reflecting a particular aspect of the system. For more information see [Sys06, FLOQ11].

In addition to the content of these UML diagrams, the UML profile of Syspect also determines supplementary properties of the stereotypes *capsule* and *data*. According to the UML standard [OMG09], such properties can be defined with attributes corresponding to a stereotype that are called *tags* [MORW08]. Table 7.1 lists all tags that correspond to the operations in capsules and data classes and Table 7.2 all tags belonging directly to a capsule or a data class. For instance, the tag *init* is used to specify a constraint that describes the initial condition of a capsule in terms of a Z predicate.

In Syspect, these tags are specified using user-friendly dialogs and textual editors with support for the input of Z predicates. Figure 7.4 gives an idea of the user interface by picturing a class diagram, where the RBC-Controller is selected, together with a corresponding property view, where the *init* tag, i.e., the initial constraint, of the controller can be specified with a Z predicate. The input of those Z predicates is

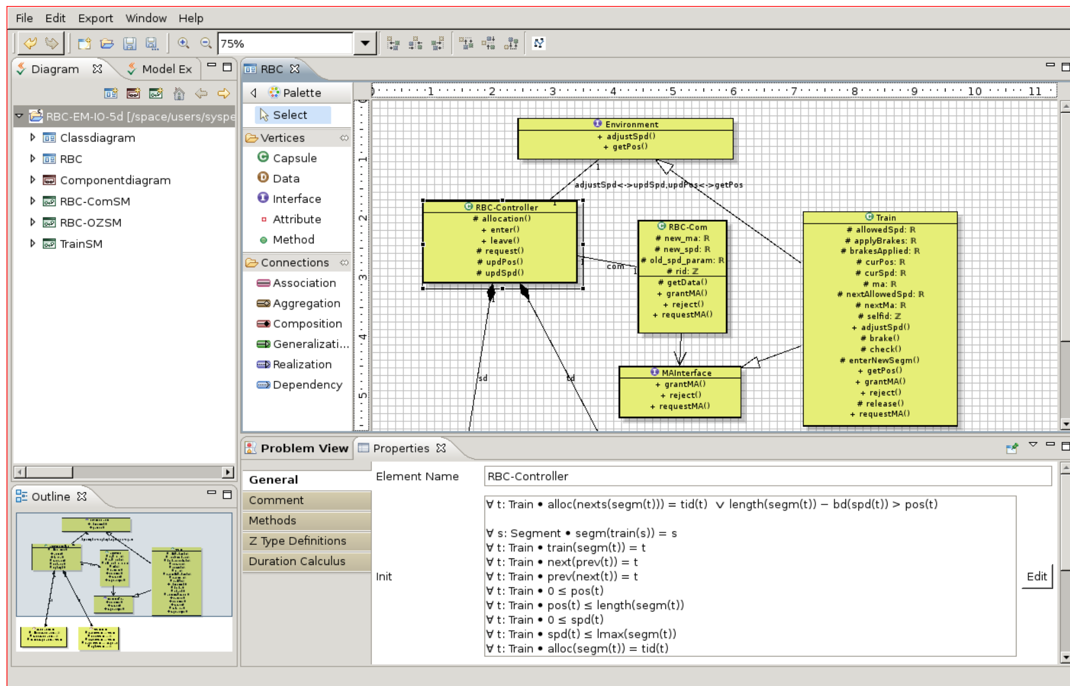


Figure 7.4.: Screenshot of Syspect

either done by selecting the corresponding predicate parts via specific buttons or by writing LaTeX commands for Z [ISO02] that are replaced on-the-fly by Z symbols.[1] The latter is very convenient for users familiar with the common LaTeX representation of Z.

7.1.2. Tool Structure

The Syspect implementation is based on the Eclipse *Rich Client Platform* (RCP). Eclipse is an expansible software development framework structured by plug-ins. It is written in the programming language Java and it is available at

`http://www.eclipse.org`.

The RCP is the minimal set of Eclipse plug-ins that are necessary to build a rich client application using the Eclipse framework.

Being such a rich client application, Syspect takes advantage from many build-in features of Eclipse, e.g., graphical user interface (GUI) features like layered dialogues called *wizards*, SWT (Standard Widget Toolkit) widgets, a user-arrangeable GUI,

[1]Syspect internally uses the *Community Z Tools (CZT)* to handle Z expressions. For more information on CZT see `http://czt.sourceforge.net/` and [MU05].

and support for drag and drop. Moreover, Syspect inherits the extendability from Eclipse by which Syspect has been improved with a lot of verification plug-ins after its initial release. Syspect is highly configurable since plug-ins can be activated or deactivated such that different Syspect-flavors can be delivered that are tailored to the particular needs of a user. Some important plug-ins extending the core features of Syspect are mentioned in the next section.

7.1.3. Syspect Plug-Ins

Most of the existing Syspect plug-ins are export plug-ins that allow for further processing of a Syspect model. This is of course essential for the practical usage of Syspect, because such plug-ins connect Syspect with other existing tools, e.g., model checkers.

XML export. Syspect supports exporting specifications in terms of XML-files. Either the CSP-OZ-DC representation of a Syspect model can be output as XML or the CSP-OZ-DC specification is firstly translated into a PEA network, which is written into an XML file. These XML representations of Syspect models can be used as exchange format, which is for instance done in the AVACS project.

LaTeX export. Very important is the LaTeX export of Syspect. Using this plug-in a CSP-OZ-DC specification of a Syspect model is output as LaTeX file, which is the standard format to describe CSP-OZ-DC specifications. By this, Syspect provides a user-friendly graphical way to specify a system, which can then be viewed as a CSP-OZ-DC specification. This is more comfortable than writing CSP-OZ-DC specifications directly.

Image export. The diagrams of Syspect can also be exported as common image files (PDF, JPEG, PNG, EPS). The figures of diagrams in this chapter are direct exports of Syspect diagrams (Fig. 7.1 up to Fig. 7.3).

PEA editor. In [Pet09], an editor plug-in for PEA has been developed. This enables a user to model directly in terms of PEA within Syspect. Moreover, a Syspect model can be exported to PEA which is then displayed in the PEA editor. This is useful for debugging Syspect specifications, because the semantical PEA representation of all Syspect elements, particularly for the DC formulae, can be examined and modified.

Furthermore, there are several verification plug-ins, which are explained in the next section.

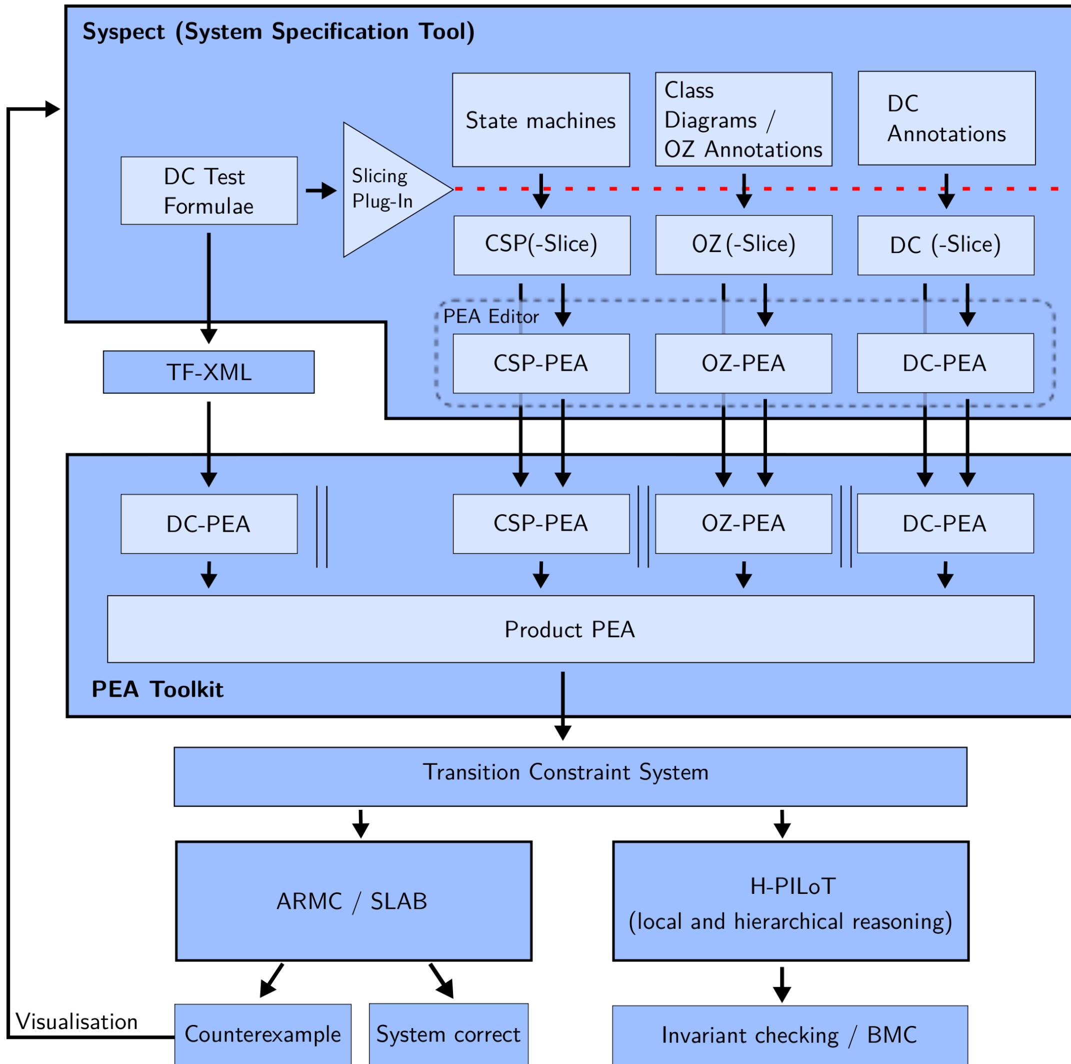


Figure 7.5.: Syspect verification tool chain (modified version of a figure in [Brü08b])

7.2. Verification with Syspect

A major goal of Syspect is to provide access to formal verification approaches through suitable plug-ins. Particularly, the CSP-OZ-DC verification approach that is relevant in the context of this work is supported by Syspect. An overview of the tool chain for Syspect verification is given in Fig. 7.5.

Since every model in the UML profile can be transformed into a CSP-OZ-DC specification, which in turn has a semantics in terms of PEA and Transition Constraint Systems (TCS) [Hoe06], the main approach in Syspect is to employ verification tools that use TCS as their input language. Additionally, there is an implementation that uses the CSP semantics for CSP-OZ [Fis00], by what verification with the FDR model checker for CSP is possible. The remaining chapter focusses on the TCS verification approach and decompositional verification techniques that are implemented in Syspect to simplify verification tasks.

7.2.1. Transition Constraint Systems

Figure 7.5 shows the translation steps and tools involved in the translation to TCS. As explained in Sect. 7.1.1, the UML diagrams of Syspect have via a UML profile a direct correspondence to CSP-OZ-DC elements. The translation from UML to CSP-OZ-DC [Sys06, MORW08] proceeds as follows:

- Capsules are translated into CSP-OZ-DC classes. Attributes and methods of capsules directly become attributes and methods of the class. This is possible, because the methods of capsules are already in Z syntax with a dedicated list of symbols that can be changed in a method (tag *changes*) and with dedicated input and output variables (tags *in* and *out*).
- Data classes are translated into OZ classes, without control structure or timing part.
- State machines in Syspect always belong to a capsule and can be translated into equivalent CSP processes.
- DC formulae also belong to capsules and can directly be conveyed to the corresponding CSP-OZ-DC class.

Afterwards the specification is translated into PEA according to the operational semantics of CSP-OZ-DC (Sect. 2.3.2). These translation steps are automatically executed in Syspect.

Compositional Reasoning. An important consequence of the compositional semantics of CSP-OZ-DC in terms of PEA is that it permits compositional reasoning in the follwing sense: whenever a subset of PEA in a parallel composition satisfies a safety property (given by a DC formula), then also the full parallel composition does.

See Thm. 2.3.8 for a formalisation. This allows for a *cone-of-influence* verification technique: it suffices to verify a safety property for a single component to conclude safety of the entire system. For example, when verifying a timing property, it often depends only on the DC part of the specification; then, only the DC part is to be considered for the verification of the property.

Translation into TCS. In a further step, the PEA are translated into Transition Constraint Systems (TCS), transition structures that are supported by the verification tools ARMC [PR07] and SLAB [BDFW07, BDFW08, DKFW10]. A TCS is a tuple $\mathcal{T} = (Var, Init, Trans)$, where *Var* is a set of unprimed variables, *Init* is an initial constraint describing all initial states of the system, and *Trans* is a transition constraint describing state changes, where unprimed variables refer to the state before the change and primed variables to the state after the change. At the level of TCS, the clocks of PEA are represented as real-valued data variables.

The translation from a PEA to a TCS is described in [HM05b, Hoe06] and it is relatively straightforward in that the only constructs in PEA that deserve a special consideration here are the PEA locations and the clock constructs. The remaining constructs like state changes during PEA transitions can directly be treated as TCS transitions, because PEA (and likewise CSP-OZ-DC) use also a primed and unprimed constraint notation to describe state changes. PEA locations are encoded in a TCS using a program counter variable, and clocks of PEA are encoded by real-valued variables. Since clock constraints in PEA are always convex, the progress of time can be modelled using a further real-valued variable that is added to the clocks in each transition step. For details see [HM05b, Hoe06, FJSS07].

The drawback of this procedure is the state space explosion, because a system that consists of parallel components has to be translated into a single TCS (Fig. 7.5), since at present the model checkers ARMC and SLAB can process only a single TCS.

PEA Toolkit. The translation to a CSP-OZ-DC specification is executed within Syspect as depicted in Fig. 7.5. Computing the parallel product of all PEA and translating it into a TCS is done by the *PEA toolkit*[2] [MFHR08], which is part of Syspect but also available as a separate library for the handling of PEA. The translation of DC formulae into PEA is also implemented in the PEA toolkit.

7.2.2. Verification of Syspect Specifications

Verification with SLAB/ARMC. The model checker ARMC[3] [PR07], developed at the Max-Planck-Institut für Informatik in Saarbrücken, and SLAB[4] [BDFW07, DKFW10], developed at the Saarland University, both take a single TCS as input.

[2] Available at `http://csd.informatik.uni-oldenburg.de/projects/epea.html`
[3] Available at `http://www7.in.tum.de/~rybal/armc/`
[4] Available at `http://react.cs.uni-sb.de/tools/slab.html`

Transition	OZ: Train	SM: Train	DC: check
563	check: (curPos ≥ dp ∧ allowedSpd > nextAllowedSpd) ⇒ applyBrakes′ = 1 (curPos ≥ dp ∧ allowedSpd ≤ nextAllowedSpd) ⇒ applyBrakes′ = 0 curPos < dp ⇒ applyBrakes′ = 0	State (InitialState3, InitialState1, InitialState2) -- check --> State (InitialState3, State4, InitialState2)	In Phases: TRUE ⌈TRUE⌉ ∧ ⊟ check ∧ ℓ > 1 (waiting)
5699	brake: applyBrakes ≥ 1 ∧ brakesApplied′ = 1	State (InitialState3, State4, InitialState2) -- brake --> State (InitialState3, InitialState1, InitialState2)	In Phases: TRUE ⌈TRUE⌉ ∧ ⊟ check ∧ ℓ > 1 (waiting)
6393	check: (curPos ≥ dp ∧ allowedSpd > nextAllowedSpd) ⇒ applyBrakes′ = 1 (curPos ≥ dp ∧ allowedSpd ≤ nextAllowedSpd) ⇒ applyBrakes′ = 0 curPos < dp ⇒ applyBrakes′ = 0	State (InitialState3, InitialState1, InitialState2) -- check --> State (InitialState3, State4, InitialState2)	In Phases: TRUE ⌈TRUE⌉ ∧ ⊟ check ∧ ℓ > 1 (waiting)
5698	brake: applyBrakes ≥ 1 ∧ brakesApplied′ = 1	State (InitialState3, State4, InitialState2) -- brake --> State (InitialState3, InitialState1, InitialState2)	In Phases: TRUE ⌈TRUE⌉ ∧ ⊟ check ∧ ℓ > 1 (waiting)

Figure 7.6.: Counterexample view in Syspect

They verify safety properties of infinite-state systems by checking reachability of *bad* states and return counterexample traces if these bad states are reachable.

In [Mey05, MFR06, MFHR08], these bad states are specified using DC formulae that are also translated into TCS. To this end, the DC counterexample traces are extended to DC test formulae that can be used to specify bad behaviour that shall be avoided by a correct implementation. The PEA toolkit also implements the translation of test formulae into PEA such that ARMC and SLAB can be used to verify Syspect specifications against DC test formulae by translating the specification and the test formulae into PEA and passing their parallel composition to the model checker.

Using the TCS verification approach, Syspect enables *parametric verification* of real-time systems in the data and the time dimension [FJSS07]. That is, it is not necessary to give specific values for all system parameters; it suffices to constrain system parameters adequately to guarantee safety.

As explained above, all constraints occurring in PEA, and thus also the constraints of Syspect specifications, are directly shifted down to the TCS level (modulo syntactical modifications). Since Syspect and CSP-OZ-DC both allow arbitrary Z expressions in operation schemes to describe state changes, not every valid Syspect specification can be verified with ARMC or SLAB, which both can handle only linear arithmetic constraints over real-valued variables. The design decision for Syspect was not to restrict the Syspect language to a specific CSP-OZ-DC subset that can be verified but to allow an expressive language at the level of Syspect, which is also supported by CSP-OZ-DC and PEA. The advantage of this is that Syspect can be used to design specifications, even though they cannot be verified. Moreover, Syspect can easily be extended by further verification plug-ins without the need to adapt the Syspect language itself. An example for a verification plug-in that allows for more complex input constraints than SLAB or ARMC is given with the H-PILoT plug-in, described below.

Verification Feedback. Syspect supports this verification approach by allowing users to specify correctness properties of a model by DC test formulae. Using convenient input dialogues, a user may export a TCS for a given test formula that can directly be

verified with SLAB or ARMC. In addition, ARMC can be called from within Syspect. In this case, the feedback from the model checker is also displayed within Syspect. To this end, a Syspect plug-in of Ulrich Hobelmann [Hob07] maps the counterexample trace provided by ARMC back to the corresponding high-level elements of the Syspect model (cf. Fig. 7.5): a counterexample view is displayed that allows step-wise following the trace to the error. This way, a user can reproduce the origin of a system error and correct the model accordingly. The counterexample view of Syspect is pictured in Fig. 7.6.

Verification with H-PILoT. To support more complex data structures, Syspect additionally interfaces with the verification tool H-PILoT[5] [ISS09] that has been developed at the Max-Planck-Institut für Informatik in Saarbrücken. H-PILoT uses hierarchical reasoning in chains of local theory extensions to reduce the complexity of verification tasks. To this end, the satisfiability of constraints over specific theory extensions that are identified to be local are reduced to the satisfiability of constraints in a base theory for that a dedicated prover exists. Standard SMT solvers can then be used to check the satisfiability of the formulae of the base theory. With this approach, the invariant checking problem for local theory extensions becomes decidable. By this means, CSP-OZ-DC specifications with properties over rich data types like arrays [FJSS07] or pointer data structures [FIJSS10a] can be verified.

In the train example of Fig. 7.1, that is taken from [FIJSS10a, FIJSS10b], a linked list is modelled with a function *next* from *Train* to *Train*. Changes or invariants over this list are modelled with quantified expressions like

$$\forall\, t : \mathit{Train} \bullet \mathit{next}(\mathit{prev}(t)) = t.$$

The safety property that is to be verified is that the RBC controller never assigns one segment to two different trains, which can be expressed by the invariant property

$$\forall\, t_1, t_2 : \mathit{Train}.\ t_1 \neq t_2 \rightarrow \mathit{sid}(\mathit{segm}(t_1)) \neq \mathit{sid}(\mathit{segm}(t_2)).$$

With the hierarchical reasoning approach, such invariant checking problems are reduced to a decidable fragment. The advantage of having a decidable problem is that the solver returns a result for correct and incorrect verification tasks. In the case that a verification task fails, H-PILoT returns with the help of the underlying solver a model that violates the desired property. This helps the user to correct the model.

Like the ARMC/SLAB verification approach, H-PILoT verification is parametric for data and time parameters. Moreover, as H-PILoT allows for verifying complex pointer structures or arrays, it is also possible to verify a parametric number of components, e.g., an arbitrary number of trains like in [FJSS07].

[5] Available at `http://www.mpi-inf.mpg.de/~ihlemann/software/index.html`

7.2.3. Slicing CSP-OZ-DC Specifications in Syspect

Syspect is designed to cope with systems with concurrent components, real-time, and (possibly) infinite data types like lists. Due to this inherent complexity of the specifications written in Syspect, direct verification is often not possible, because of the state space explosion problem. Thus, further decomposition techniques are important to enable verification of large and realistic models.

In Sect. 6.4.1, we have introduced slicing of formal specifications as decomposition technique complementing our approach. Slicing of CSP-OZ-DC specifications is also supported by a Syspect plug-in that has been developed by Sven Linker in the context of the PhD thesis of Ingo Brückner [Brü08b]. With this plug-in, slicing can be enabled for the export and verification functions of Syspect. The user has to choose a slicing criterion, by what the reduced CSP-OZ-DC specification is automatically computed according to this criterion.

7.2.4. Further Verification Plug-Ins

Furthermore, there are several verification plug-ins for Syspect that have been developed at the University of Paderborn.

A CSP verification plug-in translates a Syspect model *without* timing constraints to a CSP-OZ [Fis00] specification. CSP-OZ has a semantics in terms of CSP_M [Sca98], a machine readable CSP dialect that can be used as input for the model checker FDR2[6] for CSP, which can directly be called from within Syspect. The property that is to be verified needs to be given as a CSP_M expression.

In [MWW08, Met10], a technique has been introduced to *decompose* parallel CSP-OZ specifications into two phases using a user-specified cut point. These phases are then verified with a learning-based *assume-guarantee* rule. This decomposition approach has also been implemented in Syspect. Applying FDR2, the plug-in can be used to verify specifications without timing constraints and infinite data.

7.3. Syspect Verification Architecture Plug-In

To enable decompositional verification of real-time systems, Syspect incorporates a Verification Architecture plug-in, supporting modelling and verification of VAs. The plug-in has initially been implemented by Matthias Peters in his diploma thesis [Pet10], and it has been developed further by the author.

7.3.1. Modelling of VAs

VAs can be specified within Syspect using an extended state machine editor, which consists basically of the standard state machine editor of Syspect. In addition, the VA editor allows to add unknown phases that are annotated with local real-time

[6] `http://www.fsel.com/software.html`

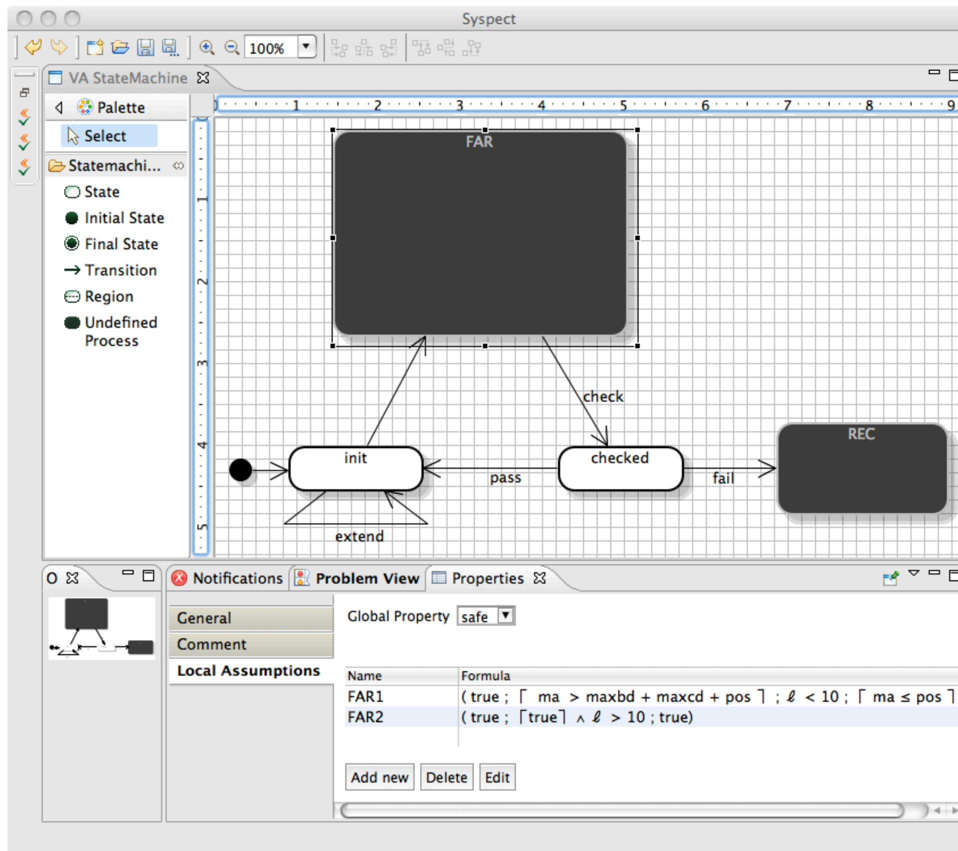


Figure 7.7.: VA of the running example modelled in Syspect

constraints in terms of DC counterexample traces. Figure 7.7 shows the VA for the running example as Syspect model. The pictured state machine represents the VA process of the running example from Sect. 3.1.4.

The plug-in makes use of the standard input dialogs of Syspect. By this means, state changes, corresponding to events, are defined by Z expression analogous to state changes described in Syspect class diagrams. Likewise the DC formulae are input in the usual way. Figure 7.7 shows, besides the VA state machine, the input dialog for DC assumptions over unknowns.

7.3.2. CSP-OZ-DC Representation of a VA

The VA model can also be exported into a CSP-OZ-DC specification in LaTeX syntax. In this CSP-OZ-DC representation of the VA, all channels of the VA process are declared in the interface and state variables in the state schema. The CSP process of the specification is the CSP process with unknown parts generated from the extended state machine. The CSP process also contains the DC assumptions for the unknown

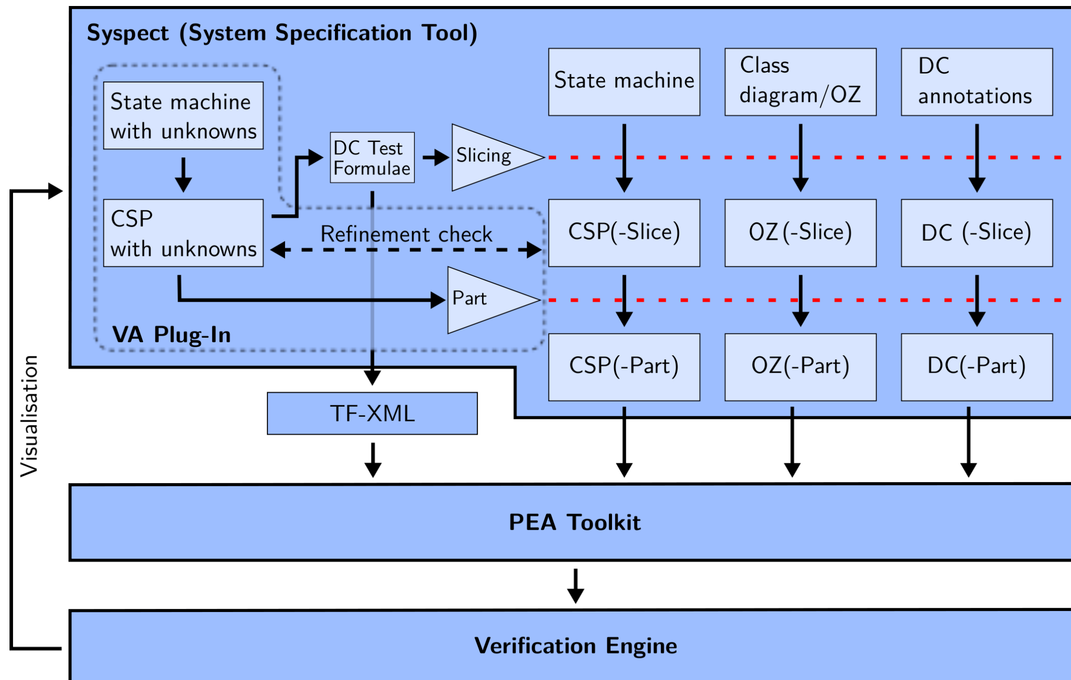


Figure 7.8.: Tool chain for VA plug-in

parts but no state changes. State changes are described by operation schemas associated with the events as in standard CSP-OZ-DC. The DC part is always empty. Appendix A.1.3 contains the VA of the running example represented as CSP-OZ-DC specification exported by Syspect.

7.3.3. Verification of VAs

Given a concrete Syspect specification and a Syspect VA, it can automatically be verified that the concrete model actually refines a VA. To this end, the refinement relation of the process structure is syntactically checked according to the rules of Chap. 5. In addition, the verification of the local real-time assumptions is also performed automatically by exporting the corresponding proof task to ARMC or SLAB.

Figure 7.8 pictures the tool chain for verifying VA instantiations. The following steps are executed:

- In a first step, the VA represented by a state machine with unknowns is translated into a CSP process with unknowns. Also, the concrete Syspect specification is translated into a CSP-OZ-DC specification.
- The structural refinement relation between the CSP process of the concrete model (not being sliced) and the VA process is syntactically checked by an algorithm of Matthias Peters, which is based on the matching rule of Def. 5.4.1.

- During the computation of the syntactical refinement, all sub-processes of the concrete model that refine an unknown process are collected.
- The user can select via a dialog which local assumptions are to be checked and whether slicing is to be used.
- For each local assumption the VA plug-in automatically slices the specification with the assumption as slicing criterion. The specification is automatically reduced to the sub-process belonging to the unknown process of the assumption. These sub-specifications are identical to the origin specification except that the CSP part of the protocol class is replaced by the sub-process, and the constraint of the initial schema is replaced by true. By this it is guaranteed that the sub-specification may start with arbitrary valuations of system variables. The possible initial values are only restricted by the assumptions on a protocol phase.
- If the specification consists of several CSP-OZ-DC classes, the relevant parallel classes for checking the current assumption are automatically selected; the initial constraints are also removed. A simple syntactical condition is checked in order to ensure that the verification with parallel components is actually admissible.
- To the resulting reduced CSP-OZ-DC specification the standard verification procedures of Syspect are applied. When using ARMC verification directly from within Syspect, a counterexample is pictured in the case that the verification fails.

Parallel components. The structural refinement check of Syspect currently supports only specifications where exactly one class implements the VA protocol (we refer to this class as *protocol class*). As already discussed in Sect. 5.6, specifications often can be reformulated such that one class implements the VA protocol. However, for convenience it is also possible to have classes running in parallel to the class implementing the VA protocol. But it is important that the restrictions of Sect. 5.6 apply to avoid that a parallel component spuriously block system runs. This particularly means that the CSP part of the parallel component is always in its initial state whenever a sub-process is entered that refines an unknown component and communicates with a parallel component. To this end, Syspect implements an (incomplete) syntactical check: it is verified that all components refining an unknown part are triggered by a unique event. All parallel components of the protocol class need to synchronise on this event and restart its CSP process when the events occur. By this it is enforced that every parallel component is actually restarted when an unknown part is entered. Syspect gives a warning if the check fails—because the user can still prove manually that the parallel components of the protocol class introduce no spurious blocking when checking them in isolation.

In addition, Syspect automatically deselects parallel classes if they are obviously independent from the current proof task. This is for instance the case if no communication with a component takes place.

Note that it is always safe to omit parallel components for CSP-OZ-DC (see Sect. 2.3.2 and 7.2.1). Thus, it is often possible to increase efficiency of the verification by deselecting components that may not be important for a proof task. But if a counterexample is returned, it may be a spurious counterexample that actually cannot occur with all parallel components enabled. Syspect provides a dialog to conveniently enable and disable components for verification.

7.4. Discussion

In this chapter, we have introduced our tool set to treat complex real-time systems with engineering-like graphical modelling techniques. The UML tool Syspect integrates UML modelling and formal methods by providing the translation into formal CSP-OZ-DC specifications. VAs are supported with a dedicated plug-in that allows modelling of VA processes with annotated state machines. The VA plug-in prototypically checks the instantiation relation between a VA and a concrete model: the structural refinement of the process structure can be established for the case that the VA process is refined by one protocol class in the concrete model; the proof tasks corresponding to the VA are automatically processed.

The class of concrete models that can be verified against a VA are currently restricted to models that syntactically coincide with the VA. It is desirable to relax this restriction in future work such that also the extended refinement rule of Def. 5.3.2 is supported. In addition, the restriction to one protocol class implementing the VA process should also be relaxed, e.g., by computing the overall CSP process of a CSP-OZ-DC specification that is compared to the VA process.

We have not yet implemented the proof calculus from Chap. 4 to verify properties of the VA itself. Ideally, an implementation of our sequent calculus would also be integrated into the Syspect environment with the disadvantage that the complete sequent calculus has to be reimplemented in Syspect. For this reason, it may be preferable to connect existing sequent calculus tools to Syspect or to translate the VA process to external tools that are extended by our sequent calculus rules. To be considered are for instance the theorem provers KeY [BHS07], KIV [HBB+05], or Isabelle[7] [NPW02]. The former seems particularly well-suited, because Platzer and Quesel have already successfully implemented a sequent calculus similarly to ours [PQ08]. On the other hand, Isabelle is a generic proof assistant designed for extendability by domain-specific logics, already including a sequent-based logic.

[7] `http://isabelle.in.tum.de/`

Related Tool Environments

Process Analysis Toolkit. The Process Analysis Toolkit (PAT)[8] [SLD08] is a tool environment for CSP#, combining CSP processes with a procedural low-level programming language (in C# syntax) and a number of real-time patterns to specify deadlocks, timed interrupts, and time-outs (see also Sect. 3.4.3). PAT implements some verification approaches that are currently not available for CSP-OZ-DC. In particular, PAT is designed for verifying LTL-X specifications with fairness. Its (explicit) model checking algorithm incorporates partial order reduction and process counter abstraction techniques as well as bounded model checking. PAT also supports a refinement check for CSP# programs and a graph-based simulation of the behaviour of the input model. In contrast to Syspect, PAT does not have a diagram-based input language and, except for the fixed real-time patterns, it does not support infinite-data verification.

Fujaba. A similar concept to Syspect is implemented in the Fujaba tool suite[9] [BGH+05]. It is also based on UML diagrams and focused to real-time system development. Real-time aspects are specified with real-time state charts [BG03], using a discrete time domain. The structure of systems is specified with class and component diagrams. The Eclipse-based Fujaba also incorporates plug-ins for simulation of systems and for hybrid systems. In contrast to our approach, the semantics of the UML diagrams is not given in terms of an intermediate combined specification language but in terms of Kripke structures [GTBF03].

Fujaba also supports a behavioural design patterns approach similarly to the VA approach [GTBF03, Gie03]. Design Pattern are specified with (real-time) state charts but without a notion of data and unknown components as in our case. Their patterns consist of a connector automaton and further (concrete) automata with additional state invariants (that are not considered to verify pattern correctness). Temporal constraints over components are not included in the patterns; real-time properties are encoded in the automata of the patterns. The patterns are verified with model checking and instantiated syntactically like in our approach. Deadlock-freedom is additionally checked for instantiating components.

The main differences to the VA approach are the usage of a discrete time domain, the lacking of data constraints in the architecture, and the non-compositional verification of the patterns: the parallel product of all components of the pattern is verified against a global property. The verification of the pattern in the VA approach is performed using the sequent calculus of Chap. 4, which is particularly well-suited for patterns that incorporate rich data types. A further difference is that the Fujaba patterns are directly defined in terms of state charts, while VAs are defined in terms of the basic process language CSP; with Syspect state machines we provide an additional front-end for the CSP language.

[8] `http://www.comp.nus.edu.sg/~pat`
[9] `http://www.fujaba.de`

8 Case Studies

Every story has got a happy end – you just have to know when you stop telling.

(A Storyteller, in Preludes and Nocturnes, *Neil Gaiman)*

8.1. Running Example: Small Train Control System 191
8.1.1. Verification of the Architecture 192
8.1.2. Instantiation by a CSP-OZ-DC Model 194
8.2. European Train Control System 196
8.2.1. Case Study Scenario: Emergencies in Train Control Systems 197
8.2.2. Previous ETCS Case Studies 198
8.2.3. VA for the ETCS Case Study 202
8.2.4. VA Verification 207
8.2.5. Instantiating the VA 214
8.2.6. Checking the Instantiation with Syspect 216
8.2.7. Discussion 220

8.1. Running Example: Small Train Control System

In this thesis, we have considered fragments of a train control system as a running example. In this section, we bring the different parts of the running example together.

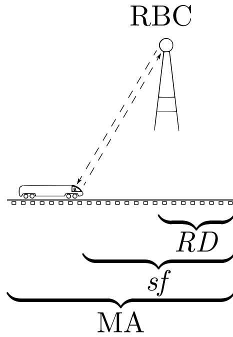


Figure 8.1.: Example

The example is motivated by the European Train Control System [ERT02] (see Sect. 1.1). The setting is pictured in Fig. 8.1: a so-called Radio Block Center (RBC) grants movement authorities (MA) to a train. The system is considered safe as long as the train stays within the MA. The distance of the train to the end of the MA is given by a real-valued variable sf, reflecting the safety of the system that shall never be below or equal to 0. As long as the train is far away from the end of the MA, we do not further restrict the behaviour of the train, but as soon as the position of the train is beyond a certain reaching distance (RD) to the end of the MA, the train has to stop before leaving its MA block. The position RD is the last position at which the train needs to apply the brakes to stop in time. The train can request extensions of MAs from the RBC at any time.

Fig. 8.2 defines this train control system as a VA. The system is described by the CSP process *System*, that consists of a choice of sub-processes.

The first process of the choice can perform an event *extend*, that extends the current MA, which is expressed in the constraint φ_{extend}. This constraint $sf' > sf$ demands that the value of the distance to the block end is increased. For realistic systems this usually means that the authority is extended, but on the level of the VA we do not exclude unrealistic behaviour like a train that jumps back to the start of the track segment. A VA usually reflects an abstract view on the system that is as general as possible but suffices to verify a concrete system.

The second process of the choice in *System* starts with an unknown process FAR that can produce arbitrary behaviour except for events from A, which are all events of this VA. It cannot change the constants of the specification, RD and CT. We further constrain the unknown process by the DC formula F_{FAR_1}, that demands that if sf is greater than RD in phase FAR, then sf cannot decrease to a value smaller than 0 within CT time units. The second formula F_{FAR_2} enforces that FAR is left after CT time units. After termination of FAR the process checks the current value of sf and, depending on that, behaves as the *System* again or it changes to a safe recovery process REC.

8.1.1. Verification of the Architecture

The safety condition we want to prove for the architecture of Fig. 8.2 is that sf never reaches 0, in terms of dCSP, $sf > RD > 0 \vdash [System]\Box sf > 0$, and we use our sequent calculus to prove its validity. To apply the proof rules for FAR and REC we need to verify the corresponding formulae F_{FAR_1}, F_{FAR_2}, and F_{REC} via the standard model checking approach for CSP-OZ-DC and DC [MFHR08]. We demonstrate this for one branch of the proof tree:

$$
\begin{array}{rcl}
\Sigma & = & (\{\mathbb{R}, \mathbb{B}, \mathit{Time}\}, \{sf : \mathbb{R}, ok : \mathbb{B}\}, \{RD : \mathbb{R}, CT : \mathit{Time}\}, \varnothing) \\
A & = & \{check, fail, pass, extend\} \\
C & = & \{RD, CT\} \\
\mathit{System} & \stackrel{c}{=} & extend \bullet \varphi_{extend} \rightarrow \mathit{System} \\
 & & \Box \\
 & & FAR \mathbin{;} check \bullet \varphi_{check} \rightarrow (fail \bullet \varphi_{fail} \rightarrow REC \,\Box\, pass \bullet \varphi_{pass} \rightarrow \mathit{System}) \\
FAR & \stackrel{c}{=} & \mathtt{Proc}_{\backslash A,C} \bullet F_{FAR_1} \wedge F_{FAR_2} \\
REC & \stackrel{c}{=} & \mathtt{Proc}^{\infty}_{\backslash A,C} \bullet F_{REC} \\
\varphi_{check} & = & \Xi(sf) \wedge sf \leq RD \wedge \neg ok' \vee \Xi(sf) \wedge sf > RD \wedge ok' \\
\varphi_{fail} & = & \Xi(sf) \wedge \neg ok \\
\varphi_{pass} & = & \Xi(sf) \wedge ok \\
\varphi_{extend} & = & sf' > sf \\
F_{FAR_1} & = & \neg\Diamond(\lceil sf > RD \rceil ^\frown \ell < CT ^\frown \lceil sf \leq 0 \rceil) \\
F_{FAR_2} & = & \neg\Diamond(\ell > CT) \\
F_{REC} & = & \neg\Diamond(\lceil sf > 0 \rceil ^\frown \lceil sf \leq 0 \rceil)
\end{array}
$$

Figure 8.2.: VA for the train control system

$$
\dfrac{\dfrac{\dfrac{\overset{(\text{assumption}_\Box)}{sf > RD > 0 \vdash [\mathtt{Proc}_{\backslash A,C} \bullet F_{FAR}]\Box sf > 0}}{sf > RD > 0 \vdash [FAR]\Box sf > 0}\;(\text{process call}) \qquad \vdots}{sf > RD > 0 \vdash [FAR]\Box sf > 0 \wedge [FAR][check \rightarrow \cdots \rightarrow \mathtt{Skip}]\Box sf > 0}\;(\text{and right})}{sf > RD > 0 \vdash [FAR \mathbin{;} \cdots \rightarrow \mathtt{Skip}]\Box sf > 0}\;(\text{sequence}_\Box)
$$

To close the left branch of the tree (we omit the right branch indicated by dots) we apply rule (assumption$_\Box$), i.e., we need to verify the side-condition of the rule, which can be expressed by a DC formula

$$
\begin{aligned}
T_{FAR_1} := \neg(true ^\frown \lceil (sf > RD) \rceil ^\frown \ell < CT ^\frown \lceil sf \leq 0 \rceil ^\frown true) \wedge & \qquad (8.1) \\
\neg(true ^\frown \ell > CT ^\frown true) \wedge & \qquad (8.2) \\
(\lceil sf > RD \wedge RD > 0 \rceil ^\frown true) \wedge & \qquad (8.3) \\
(true ^\frown \lceil sf \leq 0 \rceil ^\frown true). & \qquad (8.4)
\end{aligned}
$$

This formula states that for all runs fulfilling the *FAR* constraints (lines (8.1) and (8.2)) and starting with $sf > RD > 0$ (8.3), the constraint $\Box sf > 0$ is true (8.4). The latter is expressed negated, because the DC formulae represents bad behaviour, which shall be identified as unsatisfiable. We verified this property automatically

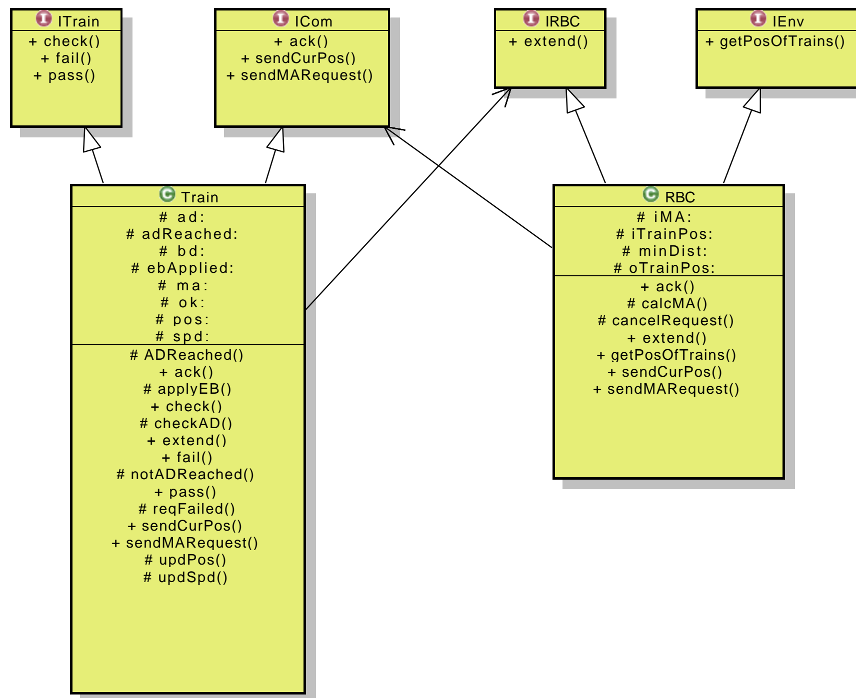


Figure 8.3.: Syspect class diagram for the train control system

within 8 seconds with ARMC using the approach of [MFHR08]. In this way, we have successfully applied our sequent calculus to verify the validity of the desired safety property. The entire proof tree can be found in Appendix A.1.1. Four external model checker calls (each finished in less than 8 seconds) to solve branches on the unknown parts *FAR* and *REC* were necessary.

8.1.2. Instantiation by a CSP-OZ-DC Model

In a second step, we have proven that a concrete CSP-OZ-DC model is an instance of the VA of Fig. 8.2. Parts of the concrete model for the train control VA have already been presented as example in Chap. 2. The model comprises two components: a train and an RBC, where the train implements the VA protocol. Figure 8.3 contains the Syspect class diagram of the case study and shows how both classes are synchronised (namely by events *extend*, *ack*, *sendCurPos*, and *sendMARequest*). The main phase of the protocol is the *FAR* phase. In this phase, the train periodically updates its position. In addition, the train cyclically checks its position. If the position is beyond an *approaching distance (ad)*, the train requests a new MA from the RBC. To this end, the train also sends position reports to the RBC. Figure 8.4 shows a part of the CSP-OZ-DC specification. The full concrete model can be found in Appendix A.1.2. In order to use ARMC or SLAB as model checkers, all system variables are modelled

Train

$\texttt{method}\ ack, check, fail, pass$
$\texttt{method}\ sendCurPos : [curPos! : \mathbb{R}]$
$\texttt{method}\ sendMARequest : [curPos! : \mathbb{R};\ reqDist! : \mathbb{R}]$
$\texttt{chan}\ extend : [newMA? : \mathbb{R}]$
$\texttt{local_chan}\ ADReached, applyEB, checkAD, notADReached, reqFailed, updPos, updSpd$

$$
\begin{array}{rcl}
main & \stackrel{c}{=} & ((extend \to main) \sqcap FAR) \\
FAR & \stackrel{c}{=} & (InitialState0 \,|||\, InitialState1 \,|||\, InitialState2) \mathbin{;} (check \to Checked) \\
InitialState0 & \stackrel{c}{=} & (updSpd \to State0) \\
InitialState1 & \stackrel{c}{=} & ((sendCurPos \to InitialState1) \sqcap \texttt{Skip}) \\
InitialState2 & \stackrel{c}{=} & ((checkAD \to State5) \sqcap \texttt{Skip}) \\
InitialState3 & \stackrel{c}{=} & (applyEB \to State8) \\
REC & \stackrel{c}{=} & (InitialState3 \mathbin{;} \texttt{Stop}) \\
State0 & \stackrel{c}{=} & (updPos \to State1) \\
State1 & \stackrel{c}{=} & (InitialState0 \sqcap \texttt{Skip}) \\
State3 & \stackrel{c}{=} & ((ack \to \texttt{Skip}) \sqcap (reqFailed \to \texttt{Skip}) \sqcap request) \\
State5 & \stackrel{c}{=} & ((ADReached \to request) \\
 & \Box & (notADReached \to InitialState2)) \\
State8 & \stackrel{c}{=} & (updSpd \to State9) \\
State9 & \stackrel{c}{=} & (updPos \to State8) \\
Checked & \stackrel{c}{=} & ((fail \to REC) \Box (pass \to main)) \\
request & \stackrel{c}{=} & (sendMARequest \to State3)
\end{array}
$$

State schema

$pos, spd, bd : \mathbb{R}$
$ma, ad, ok : \mathbb{R}$
$ebApplied, adReached : \mathbb{R}$

Init

$ad > maxcd$
$ma - pos > maxbd + maxcd$

effect_extend

$\Delta(ma)$
$newMA? : \mathbb{R}$

$ma' = newMA?$
$ma' - pos' > ma - pos$

effect_updPos

$\Delta(pos)$

$pos' = pos + spd \cdot CT$

effect_updSpd

$\Delta(spd)$

$0 \leq spd' \leq maxspd$
$0 \leq bd' \leq maxbd$
$ebApplied \geq 1$
$\quad \Rightarrow ma - pos - spd' \cdot CT > bd'$
$ebApplied \geq 1 \Rightarrow spd' = 0 \vee spd' < spd$
$ebApplied \geq 1 \Rightarrow bd' = 0 \vee bd' < bd$

effect_checkAD

$\Delta(adReached)$

$(ma - pos \leq maxbd + ad$
$\quad \wedge\ adReached' = 1) \vee$
$(ma - pos > maxbd + ad$
$\quad \wedge\ adReached' = 0)$

⋮

$\neg \Diamond (\updownarrow updPos \frown (\lceil true \rceil \wedge \ell < CT) \frown \updownarrow updPos)$
$\neg ((\lceil true \rceil \wedge \boxminus fail) \frown \not\updownarrow fail \frown (\lceil true \rceil \wedge \boxminus fail \wedge \boxminus check \wedge \ell > 10 \cdot CT) \frown true)$

Figure 8.4.: Part of the train specification

as reals.

The figure exemplarily shows the operation schemas for updating speed values, depending on the state of the emergency brake *ebApplied*, and the operation schema for *checkAD*, setting a flag *adReached* to 1 when the train is beyond the approaching distance to the end of the MA. The operations that are defined by the VA process (e.g., for *check*) reoccur in train with the same state change (not pictured in Fig. 8.4).

The system variable *ok* does occur in the CSP-OZ-DC model as well as in the VA process (with the difference that we have converted it to a real number for the verification with ARMC). However, the system variable *sf* and the constant *RD* do not occur in the model, because these symbols are instantiated by

$$sf = ma - pos$$
$$RD = maxbd + maxcd,$$

i.e., the safety value *sf* of the system is represented by the distance of the train to the end of the MA, and the reaching distance is composed of the maximal braking distance *maxbd* and the distance *maxcd*, which is the distance the train can move in one check cycle. Since the ARMC model checker can only deal with linear constraints, we instantiate the time constant CT with a concrete value.

For this instance of the train control VA direct verification was not possible (timeout after 40h) due to its complexity with 17 real-valued variables and clocks and over one hundred thousand transition constraints. Since the model is an instantiation of the VA, which was syntactically checked with Syspect, we only needed to verify the local DC formulae F_{FAR_1}, F_{FAR_2} and F_{REC} to conclude the safety of the entire system. This was done automatically with Syspect and ARMC in 82 min for F_{FAR_1}, in 13 sec for F_{FAR_2}, and less than one second for F_{REC}. Slicing was enabled for F_{FAR_1} and F_{REC}. The property F_{FAR_2} is not supported by the slicing plug-in. Direct verification of this property without slicing is successful but needs several hours. But we have made use of the Syspect feature allowing us to manually deselect components, which is possible here as F_{FAR_2} only depends on the train component.

Thus, even though the train control architecture is a rather small example, it nevertheless shows that concrete instances of the architecture are not necessarily easy to check. For the presented instance it was not directly possible. With the VA approach, we were able to successfully verify the case study by checking structural refinement and the local assumptions of the VA with Syspect and ARMC.

8.2. European Train Control System

In the AVACS sub-project R1, several case studies based on the European Train Control System standard [ECS99, ERT02] were developed [Fab05, MFR06, JSS06, FM06, FJSS07, MFHR08, FIJSS10a, FIJSS10b]. In particular, the case study presented in [MFHR08] motivated the Verification Architecture approach. Hence, we demonstrate in this section how the VA approach can be applied to this case study.

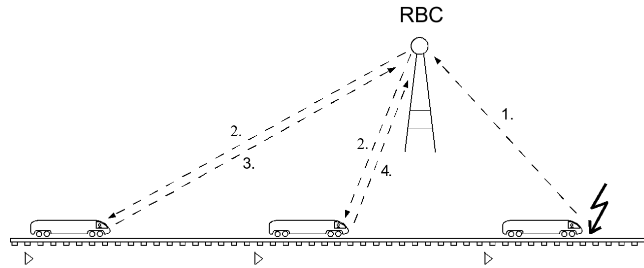


Figure 8.5.: Case study scenario

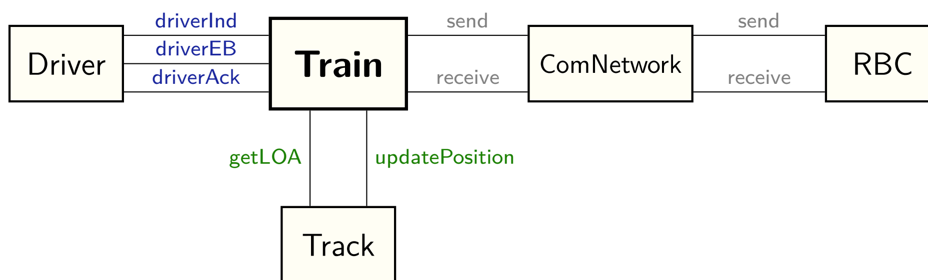


Figure 8.6.: Components of the Case Study

8.2.1. Case Study Scenario: Emergencies in Train Control Systems

The case study examines the emergency message handling in the European Train Control System (ETCS, see Sect. 1.1 for a short introduction on ETCS). The case study scenario is pictured in Fig. 8.5. We consider two consecutive trains on an (to simplify matters) infinite track segment, defined by train positions: $Pos == \mathbb{R}$. The trains measure their positions periodically and adjust their speed between a lower bound, the target speed $SysTrgSpd$ and $SysMaxSpd$, such that the follower is always able to brake safely applying the service brakes. To this end, the *service brake intervention limit* (SBI) is calculated periodically: the last position where the train has to apply the service brakes such that it remains possible to stop safely. If the first train detects an emergency, it sends an alert to the RBC which forwards this message within 5 time units. If necessary the follower immediately brakes. Otherwise, the emergency is indicated to the driver (within 1.5 time units) who has to acknowledge the warning (i.e., the driver takes responsibility for driving and the system is assumed to be safe) or to apply the emergency brakes. If the driver does not react, the emergency brakes are applied automatically by the ETCS on-board unit within 5 time units. The desired property in our case study is that the trains do never collide or that the driver overrules the warning and takes responsibility for the safety.

8.2.2. Previous ETCS Case Studies

In the previous ETCS case studies of [FM06, MFR06, MFHR08], the ETCS systems were directly modelled in terms of CSP-OZ-DC in order to adequately encompass the different aspects of the system, the control flow, data changes, and real-time aspects. We shortly summarise the basic ideas of the previous ETCS models as published in, e.g., [MFHR08].

The case study model consists of five components: *Train*, *RBC*, *Track*, *Driver*, and a communication layer *ComNetwork*, which is used to model the transfer times of messages between trains and RBC. These components are sketched in Fig. 8.6 together with their connections by channels. The main focus of the case study is on the train component incorporating the logic for the emergency message handling. The trains communicate with the RBC using the channels *send* and *receive*, which are here used to model that the RBC receives emergency alerts from the first train and timely forwards a warning to the follower. The blue channels in Fig. 8.6 are channels without parameters by which a train indicates emergency situations to the driver, who may choose to acknowledge the indication or to apply the brakes. The track represents the track-side equipment and can be seen as the environment of the case study. It periodically receives position updates from the trains and calculates a new limit of authority (LOA) if requested (*getLOA*) by the train. We abstract from the ETCS movement authority procedure (which is modelled in a simple form in Sect. 8.1), in which a LOA needs to be granted by the RBC. Instead a new LOA is calculated in the Track component in such a way that always the worst case is assumed, namely that the new LOA is closely behind the end of the preceding train. Additionally, the track component guarantees that the train positions are initially safe.

CSP-OZ-DC Specification

In the following, we show parts of the main *Train* component of the case study model. The full case study model from [MFHR08] is quoted in Appendix A.2.1.

Interface part. To simplify the verification tasks, we distinguish between the first and the second train on the track and consider two classes *FrontTrain* and *RearTrain*. The following interface declares the channels used by *RearTrain*:

$RearTrain(ID : TrainID;\ StartPosition : Position)$

$\mathtt{chan}\ send : [m! : Message, id : \{ID\}]$
$\mathtt{chan}\ receive : [m? : Message, id : \{ID\}]$
$\mathtt{chan}\ updatePosition : [id : \{ID\}, pos! : Position]$
$\mathtt{chan}\ indication$
$\mathtt{chan}\ getLOA : [id : \{ID\}, loa? : Position]$
$\mathtt{chan}\ computeSBI : [loa?, sbi! : Position]$
$\mathtt{chan}\ driverAck, driverEB : [id : \{ID\}]$
$\mathtt{local_chan}\ applySB, releaseSB, selectSpeed, applyEB$
$\mathtt{local_chan}\ getPosition : [pos! : Position]$

This interface declares the public channels used for communications between the components as depicted in Fig. 8.6. The parameter m of the channels *send* and *receive* is of type

$$Message ::= Alert \mid Warning \mid Ack.$$

In addition, there are some local channels, e.g., *applySB* and *releaseSB* for applying and releasing the service brakes, or *applyEB* for applying the emergency brake.

CSP part. The CSP part of *RearTrain* consists of an interleaving of the two subprocesses *Running* and *HandleEM* with the basic idea that the *Running* component models the movement of the train, whereas *HandleEM* models the emergency handling procedure.

$$\begin{aligned}
\texttt{main} &\overset{c}{=} Running \mid\mid\mid HandleEM \\
Running &\overset{c}{=} updatePosition.ID?pos \rightarrow getLOA.ID?loa \rightarrow \\
&\quad computeSBI!loa?sbi \rightarrow \\
&\quad \textbf{if } sbi \leq pos \\
&\quad \textbf{then } applySB \rightarrow selectSpeed \rightarrow Running \\
&\quad \textbf{else } releaseSB \rightarrow selectSpeed \rightarrow Running
\end{aligned}$$

The subprocess *Running* states that every *updatePosition* is followed by a *getLOA* event that gets the new limit of authority (LOA). The next permitted event is *computeSBI* that takes the new LOA as argument and computes the *sbi* position that is needed in the **if** statement. If the current position of the train is already beyond *sbi*, the train has to break down to *TargetSpeed*, the speed the train is allowed to run in the vicinity of the LOA according to [ERT02]. Otherwise, the train releases the brakes and selects a new speed value up to *MaxSpeed*. When the RBC sends an emergency warning, the train receives this message on the channel *receive* with the process *HandleEM*:

$$\begin{aligned}
HandleEM &\overset{c}{=} receive.Warning.ID \rightarrow send.Ack.ID \rightarrow \\
&\qquad getPosition?pos \rightarrow getSBI?sbi \rightarrow \\
&\qquad \textbf{if } pos < sbi - OffsetDistance \\
&\qquad \textbf{then } indication.ID \rightarrow \\
&\qquad\quad (driverAck.ID \rightarrow DriverResponsible \\
&\qquad\quad\ \ \square\ EmergencyStop \\
&\qquad\quad\ \ \square\ driverEB.ID \rightarrow EmergencyStop) \\
&\qquad \textbf{else } EmergencyStop \\
EmergencyStop &\overset{c}{=} applyEB \rightarrow \texttt{Skip} \\
DriverResponsible &\overset{c}{=} \texttt{Skip}
\end{aligned}$$

Depending on the current distance to the SBI position, either an emergency stop is immediately initiated, or the emergency situation is firstly indicated to the driver if the distance to the SBI position is larger than *OffsetDistance*.

OZ part. The OZ part of *RearTrain* defines attributes of the class, e.g., for the current position of the train (*currentPosition*), the current braking mode, or the position *standstillEB* that will be reached by the train if applying the emergency brakes.

$currentPosition, currentSpeed : Position$
$sbi, standstillEB : Position$
$brakingMode : BrakingMode$

`Init`
$brakingMode = None$
$TargetSpeed < currentSpeed \leq MaxSpeed$
$currentPosition = StartPosition$

To simplify the treatment of units, we measure speed in terms of $\frac{Position}{updateBound}$, where *updateBound* is the time between two position updates. The `Init` schema specifies that the train is initially not braking, the speed is between *TargetSpeed* and *MaxSpeed*, and the current position is set to *StartPosition*. Note that *TargetSpeed* and *MaxSpeed*, as well as *Length*, *TargetSpeedDistance*, and *StopDistance* are global constants defined outside the classes.

As example for operation schemas of *RearTrain* we consider the schema for *selectSpeed*.

`com_`*selectSpeed*
$\Delta(currentSpeed)$

if $brakingMode = None$ **then**
  $TargetSpeed < currentSpeed' \leq MaxSpeed$
if $brakingMode = ServiceBrake$ **then**
  $currentSpeed' = TargetSpeed$
if $brakingMode = EmergencyBrake$ **then**
  $(currentSpeed' < TargetSpeed \ \wedge$
  $currentPosition + currentSpeed' \leq standstillEB)$
$\vee \ (currentSpeed' = 0 \ \wedge$
  $currentPosition + TargetSpeed > standstillEB)$

The idea for constraining `com_`*selectSpeed* is as follows. If the train is not braking, it selects a new speed value between *TargetSpeed* and *MaxSpeed*. If the service brakes are applied, the new speed is set to *TargetSpeed*, and if the emergency brakes are applied, the speed is set to a value below *TargetSpeed*. In the latter case, we take the *standstillEB* position into account, which is computed in `com_`*applyEB*.

DC part. Finally, the timing constraints of *RearTrain* are given by the following DC formulae:

$$
\begin{array}{l}
\neg\Diamond(\updownarrow updatePosition ^\frown updateBound > \ell ^\frown \updownarrow updatePosition) \\
\neg\Diamond(\boxminus updatePosition \wedge updateBound < \ell) \\
\neg\Diamond(\updownarrow receive.Warning.ID ^\frown \boxminus applyEB \wedge \boxminus indication \wedge 0.5 < \ell \\
\qquad ^\frown \boxminus indication \wedge 1 < \ell) \\
\neg\Diamond(\updownarrow indication ^\frown \boxminus driverAck \wedge \boxminus applyEB \wedge 5 < \ell)
\end{array}
$$

The first two DC formulae in the train specification define that *updatePosition* is a periodic event occurring every *updateBound* time units. The third DC formula demands that after receiving an emergency warning, the emergency brakes are applied within 0.5 time units, or the emergency is indicated to the driver within 1.5 time units. The driver has to acknowledge the indication, or the emergency brakes are automatically applied within 5 time units, which is required by the last formula.

Leading Train. The *LeadingTrain* component is a simplified version of the *RearTrain* class. The CSP part only considers the running behaviour as well as the detection of emergencies. If an emergency is detected, we assume the worst case, i.e., the train stops immediately.

Verification of the Case Study

The desired safety property in our case study is that the trains will never collide if the driver has not overruled the warning of the system. In the latter case, the system is assumed to be safe. In the original ETCS case study this property has been expressed with the DC formula

$$
\neg(\ell > 0 \wedge \boxminus driverAck ^\frown \lceil RearTrain.currentPosition > LeadingTrain.currentPosition - Length \rceil). \quad (8.5)
$$

In [MFHR08], this property and the CSP-OZ-DC specification were translated into PEA according to procedure shortly described in Sect. 2.3.2. This translation results in 18 parallel PEA with 10 real-valued state variables, 15 parameters of channels, and 9 continuous-time clocks. Hence, this model is too large to verify the global safety property (8.5) without decomposition techniques.

As it is not possible to directly verify the collision freedom of the trains in a single step, the safety property was manually decomposed into smaller parts. Even though the case study has not been modelled with a focus to a protocol-based verification, it turned out that the emergency message protocol of the case study naturally leads to a composition of the safety property that can be used to simplify its verification. The following phases of the protocol have been identified:

1. Without emergency detection there is always a (safe) minimal distance between the trains.
2. After an emergency detection the follower starts braking within 8 time units.
3. Starting with the minimal distance of 1., the rear train still has a distance of *StopDistance* to the preceding train after 8 time units.
4. If the train applies the emergency brakes with a distance of *StopDistance* to the the preceding train, it always stops before reaching the danger point.

These properties were formalised in [MFHR08] using DC formulae. Then, ARMC was used to verify that each of them holds for specific parts of the CSP-OZ-DC model. By this means, the application of the DC model checking approach for CSP-OZ-DC specifications has been demonstrated. It has not been formally proven that the properties, which are locally valid for parts of the CSP-OZ-DC specification, actually imply the desired global property.

8.2.3. VA for the ETCS Case Study

The Verification Architecture approach has been developed in order to provide a formal framework for protocol-based verification of real-time systems. Hence, we show in this section how the VA approach can be used to formally verify that the manual, protocol-based splitting of the global property of the ETCS case study into local properties actually leads to a correct decomposition. To this end, we introduce and verify a VA protocol of the ETCS case study in this and in the following section. Afterwards, we present an instantiation of the VA as a Syspect model.

Signature

The signature, displayed in Fig. 8.7, comprises the symbols and types used in the architecture process. It contains, besides $\mathbb{B}$, types for position values and for speed values and a type for the braking mode:

$$BrkMd ::= BrkMdNone \mid BrkMdSrvBrk \mid BrkMdEmBrk.$$

Moreover, some of the symbols from the previous ETCS case study are declared as system variables. Position and speed values are stored in *LTcurPos*, *RTcurPos*, and *RTcurSpd*. The prefix *LT* stands for leading train, whereas *RT* represents the rear train. *RTstndEB* contains the standstill position of the rear train and *RTdrvRes* is Boolean flag, indicating that the driver has taken responsibility for stopping the train safely.

Also like for the original case study, there are constants for the maximal length of the train (*SysLength*), the maximal speed (*SysMaxSpd*), the target speed (*SysTrgSpd*), as well as the maximal braking distances for braking down to target speed or to a complete stop (*SysTrgSpdDst*, *SysStpDst*).

$$\Sigma = (\{Pos, Spd, BrkMd, \mathbb{B}\}, ETCSVars, ETCSConst, \varnothing)$$

$$\begin{aligned} ETCSVars = \{&RTcurSpd : Spd, \\ &RTcurPos : Pos, \\ &RTstndEB : Pos, \\ &RTbrkMd : BrkMd, \\ &RTdrvRes : \mathbb{B}, \\ <curPos : Pos\} \end{aligned} \qquad \begin{aligned} ETCSConst = \{&SysLength : Pos, \\ &SysMaxSpd : Spd, \\ &SysTrgSpd : Spd, \\ &SysTrgSpdDst : Pos, \\ &SysStpDst : Pos\} \end{aligned}$$

Figure 8.7.: Signature for the ETCS VA

VA Process

The process ETCS-EM defining the VA for the ETCS case study is given in Fig. 8.8. A graphical representation of the VA process, which we developed with the Syspect tool, is pictured in Fig. 8.9. The figure illustrates that the VA process consists mainly of four unknown processes, which are pictured as dark square nodes. These four unknown processes correspond to the four protocol phases from Sect. 8.2.2 that are used for the verification of the ETCS case study in [MFHR08]. The first unknown part *ProcWaitEM* represents the protocol phase 1 in that no emergency has been detected. The parallel unknown processes *ProcRun* and *ProcAckTime* model the phases 2 and 3 after the emergency detection: the emergency handling has to be executed in a fixed time (*ProcAckTime*, phase 2) and after this fixed time-interval the distance from the rear train to the front train is still sufficient to stop safely. The last protocol phase is realised with the unknown process *ProcRec*, representing a recovery phase.

This process structure is reflected in the VA process of Fig. 8.8: ETCS-EM begins with the unknown process of the first phase, followed by the parallel composition of sub-processes *HandleEM* and *Running*, and finally, by the recovery phase *ProcRec*. The processes *HandleEM* and *Running* synchronise[1] on the events from A_2: the event for emergency detection *LTdetEM*, *RTappEB* by which the rear train applies its emergency brakes, and *RTsetRes* for setting the driver responsibility. The sub-process *Running* ensures that the *LTdetEM* event is the first event in the phase of *ProcAckTime* and that this phase is concluded by either *RTsetRes* or *RTappEB*. Note that exactly these events are not allowed in the unknown process in *ProcRun* due to the exclusion of events from A_2.

In the process of Fig. 8.8 it is also specified which variables are not allowed to change

[1] In the following, *Running* and *HandleEM* are always synchronised on A_2. Hence, we omit the synchronisation alphabet from now on.

$$
\begin{aligned}
\text{ETCS-EM} &\overset{c}{=} \mathit{ProcWaitEM} \fatsemi \mathit{Emergency} \fatsemi \mathit{ProcRec} \\
\mathit{Emergency} &\overset{c}{=} \mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running} \\
\mathit{Running} &\overset{c}{=} (\mathit{LTdetEM} \bullet \Xi \mathit{ETCSVars} \rightarrow \mathit{ProcRun}) \fatsemi \\
&\quad ((\mathit{RTappEB} \bullet \varphi_1 \rightarrow \texttt{Skip}) \mathrel{\square} (\mathit{RTsetRes} \bullet \varphi_2 \rightarrow \texttt{Skip})) \\
\mathit{HandleEM} &\overset{c}{=} \mathit{ProcAckTime} \\
\mathit{ProcWaitEM} &\overset{c}{=} \texttt{Proc}_{\setminus A_0, \varnothing} \bullet \mathit{WaitEM} \\
\mathit{ProcRun} &\overset{c}{=} \texttt{Proc}_{\setminus A_2, V_2} \bullet \mathit{Run1} \wedge \mathit{Run2} \\
\mathit{ProcAckTime} &\overset{c}{=} \texttt{Proc}_{\setminus \varnothing, V_1} \bullet \mathit{AckTime} \\
\mathit{ProcRec} &\overset{c}{=} (\texttt{Proc}_{\setminus \varnothing, V_3} \bullet \mathit{Rec} \fatsemi \texttt{Stop}) \\
A_0 &= \{\mathit{LTdetEM}\} \\
V_1 &= \{\mathit{LTcurPos}, \mathit{RTcurPos}\} \\
A_2 &= \{\mathit{LTdetEM}, \mathit{RTappEB}, \mathit{RTsetRes}\} \\
V_2 &= \{\mathit{RTstndEB}, \mathit{RTdrvRes}\} \\
V_3 &= \{\mathit{RTstndEB}, \mathit{RTdrvRes}\} \\
\varphi_1 &= \Xi(\mathit{ETCSVars} \setminus \{\mathit{RTbrkMd}, \mathit{RTstndEB}\}) \\
&\quad \wedge \mathit{RTbrkMd}' = \mathit{BrkMdEmBrk} \\
&\quad \wedge \big(\mathit{RTcurSpd} \leq \mathit{SysTrgSpd} \\
&\qquad \Rightarrow \mathit{RTstndEB}' = \mathit{RTcurPos} + \mathit{SysStpDst}\big) \\
&\quad \wedge \big(\mathit{RTcurSpd} > \mathit{SysTrgSpd} \\
&\qquad \Rightarrow \mathit{RTstndEB}' = \mathit{RTcurPos} + \mathit{SysTrgSpdDst} + \mathit{SysStpDst}\big) \\
\varphi_2 &= \Xi(\mathit{ETCSVars} \setminus \{\mathit{RTsetRes}\}) \wedge \mathit{RTdrvRes}'
\end{aligned}
$$

Figure 8.8.: The VA process ETCS-EM for the ETCS case study

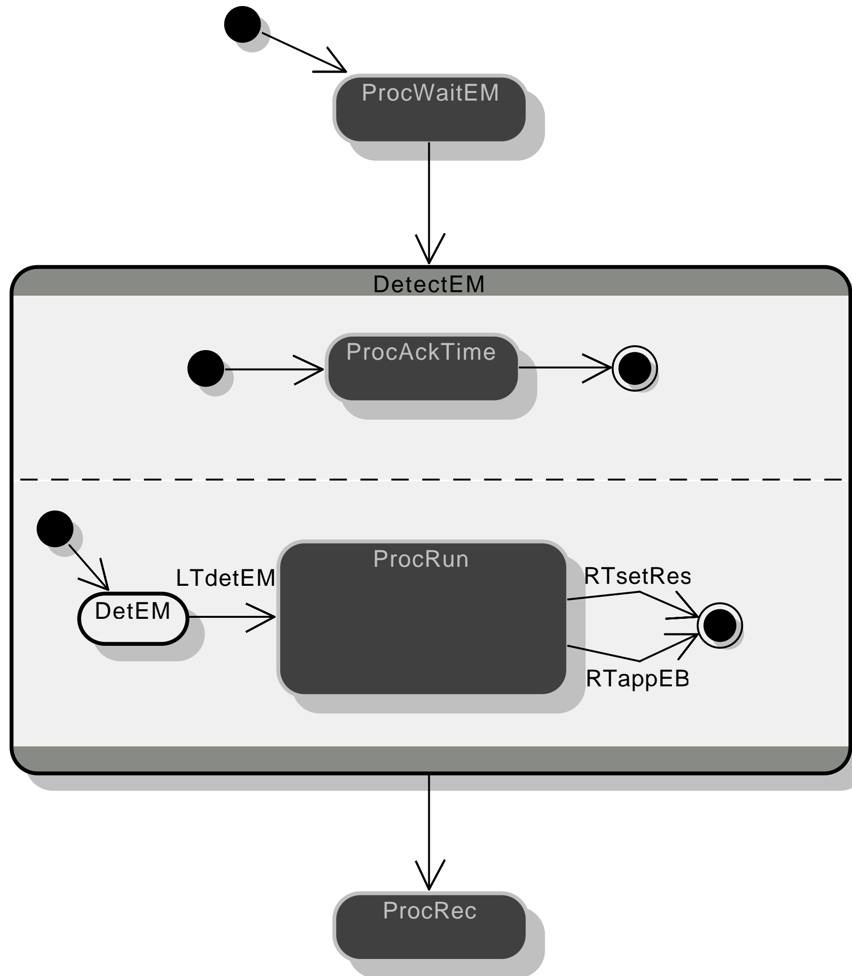


Figure 8.9.: The VA process as modelled in Syspect

during executions of the unknown parts. For instance, in *ProcAckTime* the variables from V_1, *LTcurPos* and *RTcurPos*, cannot be changed, because this sub-process is meant to deal with the emergency procedure and not the train movement.

The constraints φ_1 and φ_2 define the state change of *RTappEB* and *RTsetRes*, respectively. The latter simply sets the Boolean variable *RTdrvRes*, while the former sets the braking mode to the emergency brake and calculates the position at which the train will stop.

DC Assumption for the ETCS VA

We define the constraints over the unknown processes of Fig. 8.8 by DC counter-example traces. These DC formulae are pictured in Fig. 8.10. The formula *WaitEM* specifies that after *Init*, where *Init* is an arbitrary initial condition, the distance

$$\begin{aligned}WaitEM = \neg(\lceil Init\rceil ^\frown true ^\frown \lceil LTcurPos < RTcurPos + SysLength\\ + SysTrgSpdDst + SysStpDst\rceil ^\frown true)\end{aligned}$$

$$\begin{aligned}AckTime = \neg(\lceil true\rceil ^\frown \updownarrow LTdetEM\\ ^\frown(\boxminus RTappEB \wedge \boxminus RTdrvAck \wedge (\ell > 8))^\frown true)\end{aligned}$$

$$\begin{aligned}Run1 = \neg(\lceil LTcurPos \geq RTcurPos + SysLength + SysTrgSpdDst + SysStpDst\rceil\\ ^\frown(\lceil true\rceil \wedge (\ell < 8))\\ ^\frown\lceil LTcurPos < RTcurPos + SysLength + SysStpDst\rceil ^\frown true)\end{aligned}$$

$$\begin{aligned}Run2 = \neg(\lceil LTcurPos \geq RTcurPos + SysLength + SysTrgSpdDst + SysStpDst\rceil\\ ^\frown true ^\frown \lceil LTcurPos < RTcurPos + SysLength\\ + SysTrgSpdDst + SysStpDst \wedge RTcurSpd > SysTrgSpd\rceil ^\frown true)\end{aligned}$$

$$\begin{aligned}Rec = \neg\Diamond(\lceil RTstndEB \geq RTcurPos + SysStpDst \wedge RTbrkMd = BrkMdEmBrk\\ \wedge LTcurPos \geq RTstndEB + SysLength\rceil ^\frown true\\ ^\frown\lceil RTstndEB < RTcurPos \vee LTcurPos < RTstndEB + SysLength\rceil)\end{aligned}$$

Figure 8.10.: DC assumptions on the protocol phases

between the position of leading train *LTcurPos* and the following train *RTcurPos* is never below a certain safety distance, namely

$$SysLength + SysTrgSpdDst + SysStpDst \tag{8.6}$$

Property *AckTime* defines a maximal reaction time on *LTdetEM*, *RTappEB*, and *RTdrvAck*: after the occurrence of an *LTdetEM* event, within 8 time units either *RTappEB* or *RTdrvAck* are to be fired. *Run*1 states that if the minimal distance between the trains is at least (8.6), then for 8 time units at least a distance of *StopDistance* between the trains is preserved. *Run*1 ensures that the speed is kept above *SysTrgSpd*, as long as the distance between the trains is greater than (8.6). This is necessary in order to guarantee that a correct standstill position is calculated in *RTappEB*. *Rec* specifies that if the distance between *RTstndEB* and *RTcurPos* is greater than *SysStpDst*, the emergency braking mode is applied, and the rear end of

the first train $LTcurPos - SysLength$ is larger than $RTstndEB$, then $RTstndEB$ always separates $RTcurPos$ and $LTcurPos$.

8.2.4. VA Verification

We show with the dCSP approach presented in this work that the so-defined VA for consecutive trains actually guarantees collision freedom.

The correctness property we want to show is that the position of the second train is never beyond the rear end of the front train, or the driver has taken responsibility for keeping the system safe:

$$safe = RTcurPos < LTcurPos - SysLength \vee RTdrvRes. \tag{8.7}$$

Thus, we have to prove with the calculus from Sect. 4.3 that the dCSP formula

$$Cons, Init \vdash [\text{ETCS-EM}]\Box safe \tag{8.8}$$

holds, where $Cons$ is a condition over the global constants of the system,

$$Cons = SysLength > 0 \wedge SysTrgSpdDst > 0 \wedge SysStpDst > 0,$$

and $Init$ is the initial constraint from the DC assumption $WaitEM$. A necessary condition for $Init$ is the initial safety of the system:

$$Init \Rightarrow LTcurPos \geq RTcurPos + SysLength + SysTrgSpdDst + SysStpDst.$$

Proof Tree for Collision Freedom of the ETCS VA

The full proof tree, by which we prove collision freedom of the VA process ETCS-EM, can be found in Appendix A.2.2. Here, we present the most interesting parts of the proof tree; particularly, how we solve the parallel composition of unknown processes in the VA.

Since the full proof with all intermediate steps is longish and not very readable, we use several abbreviations in the presentation of the proof tree. First, we abbreviate some of the formulae:

$$\begin{aligned} sf &= RTcurPos + SysLength + SysTrgSpdDst + SysStpDst \\ rd &= RTcurPos + SysLength + SysStpDst \\ sf_{new} &= RTcurPos_{new} + SysLength + SysTrgSpdDst + SysStpDst \\ rd_{new} &= RTcurPos_{new} + SysLength + SysStpDst. \end{aligned}$$

Second, we apply some of the rules implicitly. In particular, we waive the explicit application of the rules (weakening left) and (weakening right), i.e., we directly omit

unnecessary formulae from a sequent, and predominantly apply (process call) implicitly. Moreover, we define the abbreviation (next-step), which represents the following sequence of rule applications:

$$\text{(next-step)} = (\text{prefix}_\delta)(\text{box step}_\delta)(\text{all right})(\text{implication right})(\text{and left})(\text{or right})$$

With this, the correctness of the VA is proven in the following way, where we apply the rules from bottom to top.

$$\dfrac{(\text{seq.}_\Box)\ \dfrac{\overset{(\text{assumption}_\Box)}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM}]\Box \mathit{safe}} \qquad (1)}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [(\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM} \fatsemi \mathit{Emergency})]\Box \mathit{safe}}}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\text{ETCS-EM}]\Box \mathit{safe}}\ (\text{process call})$$

To resolve the unknown process $\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM}$ the *oracle* rule (assumption$_\Box$) has to be applied. We show in one of the following sections that the side-condition of the rule can be proven with DC model checking. Similarly, to close sub-goal (1), rule (box assumption$_\delta$) has to be applied in order to get rid of the unknown process:

$$\dfrac{\dfrac{\dfrac{(2) \qquad (3)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [(\mathit{HandleEM} \parallel \mathit{Running}) \fatsemi \mathit{ProcRec}]\Box \mathit{safe}}\ (\text{sequence}_\Box)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{Emergency}]\Box \mathit{safe}}\ (\text{process call})}{\underset{(1)}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM}][\mathit{Emergency}]\Box \mathit{safe}}}\ (\text{box assumption}_\delta)$$

The last proof step splits the proof task into two sub-goals (2) and (3), in which we need to show that the desired property holds everywhere on (*HandleEM* ∥ *Running*) and that after termination of (*HandleEM* ∥ *Running*) every execution of the subsequent process *ProcRec* is safe:

$$\dfrac{\begin{array}{c}\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash \\ [\mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running})][\mathit{ProcRec}]\Box \mathit{safe}\end{array}}{\begin{array}{c}\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{RTcurPos} + \mathit{SysLength} + \mathit{SysTrgSpdDst} + \mathit{SysStpDst} \vdash \\ [\mathit{HandleEM} \parallel \mathit{Running}][\mathit{ProcRec}]\Box \mathit{safe} \\ (3)\end{array}}\ (\text{parallel uproc})$$

$$\dfrac{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running})]\Box \mathit{safe}}{\underset{(2)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{HandleEM} \parallel \mathit{Running}]\Box \mathit{safe}}}\ (\text{parallel uproc})$$

In both of these sub-trees for tasks (2) and (3), we make use of rule (parallel uproc) from Sect. 6.1. By this means, we replace the parallel composition that contains

unknown processes with timing constraints by an untimed process *ProcFree* with the same untimed interpretations as the original parallel composition. We demonstrate in the following section how *ProcFree* is computed and how the desired property is established for this process. But beforehand, we close the remaining sub-goal of the proof tree.

The property ensuring safe execution of *ProcRec* is given by

$$\psi = \mathit{RTdrvRes} \vee \mathit{RTbrkMd} = \mathit{BrkMdEmBrk} \wedge \mathit{RTstndEB} \leq \mathit{LTcurPos} - \mathit{Length} \\ \wedge\ \mathit{RTcurPos} \leq \mathit{RTstndEB} - \mathit{SysStpDst}.$$

That is, given that this ψ is actually satisfied after $\mathit{HandleEM} \parallel \mathit{Running}$, we can close the proof in the following way:

$$\dfrac{\dfrac{(\text{assumption}_\Box)}{\mathit{Cons}, \psi \vdash [\mathit{ProcRec}]\Box \mathit{safe}}}{\mathit{Cons}, [\mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running})]\psi \vdash [\mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running})][\mathit{ProcRec}]\Box \mathit{safe}}\ (\text{process step})$$

Resolving the Parallel Composition

To resolve the parallel composition over unknowns, we apply a translation-based approach according to Sect. 6.1.2. That is, in a first step the parallel composition containing unknowns is translated into its PEA-based semantics. To this end, the construction of page 130 is applied, resulting in a PEA network. Then, the parallel product has to be computed.

Finally, the region construction could be applied to get a finite automaton (with data constraints) from the PEA. However, due to the lack of tool support for the proof calculus, we do not compute the region construction, which results in a large untimed process that is to be handled within the calculus. Instead, we follow the proposed solution without region construction from page 140. That is, the translation from [HM05a] is used to translate the PEA into a transition constraint system in such a way that all clocks from the PEA are represented as data variables of type real. Afterwards, this finite transition system is translated back into a CSP process with data.

The resulting process $\mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running})$ is pictured in Fig. 8.11, where

$$\mathit{VR} = (\mathit{LTcurPos}, \mathit{RTcurPos}, \mathit{RTcurSpd}, \mathit{RTdrvRes}, \mathit{RTstndEB}, \mathit{RTbrkMd}).$$

By inspecting the process, one may notice that its structure is indeed as expected: The first event is *LTdetEM* as demanded; at the end of the execution an event *RTappEB* or *RTsetRes* occurs, and at the intermediate transitions the important system variables are either not changed ($\Xi\, \mathit{VR}$), or the desired properties are explicitly contained

ProcFree(*HandleEM* ∥ *Running*):

$$\begin{array}{rcl}
P & = & \mathit{ProcFree}(\mathit{HandleEM} \parallel \mathit{Running}) \\
P & \stackrel{c}{=} & \mathit{LTdetEM} \bullet \Xi \mathit{VR} \wedge x' < 8 \wedge x' = \mathit{len} \wedge \mathit{len} > 0 \rightarrow P_0 \\
P_0 & \stackrel{c}{=} & P_1 \Box P_2 \\
P_1 & \stackrel{c}{=} & P_6 \Box (a \bullet \Xi \mathit{VR} \wedge \mathit{LTcurPos}' \geq \mathit{sf}' \wedge \mathit{LTcurPos}' \geq \mathit{rd}' \\
 & & \quad \wedge x' \leq 8 \wedge x' = x + \mathit{len} \wedge \mathit{len} > 0) \rightarrow P_3 \\
P_3 & \stackrel{c}{=} & P_6 \Box \big(b \bullet \Xi(\mathit{RTstndEB}, \mathit{RTdrvAck}) \wedge \mathit{LTcurPos}' < \mathit{sf}' \wedge \mathit{LTcurPos}' \geq \mathit{rd}' \\
 & & \quad \wedge (\mathit{RTcurSpd}' \leq \mathit{SysTrgSpd} \vee \mathit{LTcurPos}' \geq \mathit{sf}') \\
 & & \quad \wedge y' = \mathit{len} \wedge y \leq 8 \wedge x' \leq 8 \wedge x' = x + \mathit{len} \wedge \mathit{len} > 0\big) \rightarrow P_4 \\
P_4 & \stackrel{c}{=} & P_5 \Box P_6 \\
P_5 & \stackrel{c}{=} & (c \bullet y' = y + \mathit{len} \wedge y' \geq 8 \wedge x' \leq 8 \wedge x' = x + \mathit{len} \wedge \mathit{len} > 0) \rightarrow P_7 \\
P_6 & \stackrel{c}{=} & d \bullet \Xi \mathit{VR} \wedge x' \leq 8 \wedge x' = x + \mathit{len} \wedge \mathit{len} > 0 \rightarrow P_7 \\
P_7 & \stackrel{c}{=} & \mathit{RTappEB} \bullet \varphi_1 \rightarrow \mathtt{Skip} \Box \mathit{RTsetRes} \bullet \varphi_2 \rightarrow \mathtt{Skip} \\
P_2 & \stackrel{c}{=} & e \bullet \Xi \mathit{VR} \wedge \mathit{LTcurPos}' < \mathit{sf}' \rightarrow P_8 \\
P_8 & \stackrel{c}{=} & \ldots \mathit{unreachable} \ldots
\end{array}$$

Figure 8.11.: Representation of the parallel composition without unknown parts

ProcFree(*HandleEM* ∥ *Running*) (without timing):

$$\begin{array}{rcl}
P & \stackrel{c}{=} & \mathit{LTdetEM} \bullet \Xi \mathit{VR} \rightarrow P_0 \\
P_0 & \stackrel{c}{=} & P_1 \Box P_2 \\
P_1 & \stackrel{c}{=} & P_6 \Box a \bullet \Xi \mathit{VR} \wedge \mathit{LTcurPos}' \geq \mathit{sf}' \wedge \mathit{LTcurPos}' \geq \mathit{rd}' \rightarrow P_3 \\
P_3 & \stackrel{c}{=} & P_6 \Box b \bullet \mathit{LTcurPos}' < \mathit{sf}' \wedge \Xi(\mathit{RTstndEB}, \mathit{RTdrvAck}) \wedge \mathit{LTcurPos}' \geq \mathit{rd}' \\
 & & \quad \wedge (\mathit{RTcurSpd}' \leq \mathit{SysTrgSpd} \vee \mathit{LTcurPos}' \geq \mathit{sf}') \rightarrow P_4 \\
P_4 & \stackrel{c}{=} & P_6 \qquad \textit{(because } P_5 \textit{ unreachable)} \\
P_6 & \stackrel{c}{=} & d \bullet \Xi \mathit{VR} \rightarrow P_7 \\
P_7 & \stackrel{c}{=} & \mathit{RTappEB} \bullet \varphi_1 \rightarrow \mathtt{Skip} \Box \mathit{RTsetRes} \bullet \varphi_2 \rightarrow \mathtt{Skip} \\
P_2 & \stackrel{c}{=} & e \bullet \Xi \mathit{VR} \wedge \mathit{LTcurPos}' < \mathit{sf}' \rightarrow P_8 \\
P_8 & \stackrel{c}{=} & \ldots \mathit{unreachable} \ldots
\end{array}$$

Figure 8.12.: Representation of *ProcFree* without constraints over clocks

in the guard (e.g., $LTcurPos' \geq sf'$). One important exception is sub-process P_5: when executing the event c, then arbitrary state changes of the system variables are possible, by which the desired property is violated. But the c-event is constrained by explicit clock constraints introduced in the translation (x and y are real-valued variables and represent the clocks of the original PEA), and it will turn out that the event c is actually blocked due to this guard over x and y. The reason is that, intuitively spoken, x is started before y (x in process P, y in process P_3), both variables are afterwards increased simultaneously, and finally, the guard of c demands that y is larger or equal to x, which is not possible.

In addition, the event e in P_2 can also not occur due to its guard over the system variables. Hence, we omit the unreachable process part P_8 following P_2.

Technically, it is no problem to analyse the entire process P with the sequent calculus in a single proof, but for sake of conciseness, we split the proof into two parts. We first show that the process P without constraints over clock variables satisfies the desired property; as long as the problematic process P_5 is ignored. This modified process is given in Fig. 8.12. In a second step, we show that the event c from P_5 can actually not occur, which proves that is was correct to remove P_5.

Proof tree for the data part. Hence, to complete the proof of our desired correctness property, we show for the computed process $ProcFree(HandleEM \parallel Running)$ of Fig. 8.12 that the desired property is valid for the δ-case and the $\Box$-case:

$$Cons, LTcurPos \geq sf \vdash [ProcFree(HandleEM \parallel Running)]\Box safe \text{ and}$$
$$Cons, LTcurPos \geq sf \vdash [ProcFree(HandleEM \parallel Running)]\psi.$$

The formula ψ is defined as above, i.e., as a formula that ensures safety of the subsequent $ProcRec$ process. The proof is not difficult and very systematic, and we conjecture that a theorem prover can solve this part of the proof fully automatic, because no invariant is to be guessed. The proof tree can be found in Appendix A.2.2.

Proof tree for timing part. In the verification of the desired property above, we have ommited the timing constraints and have removed sub-process P_5. Now, we show that the timing constraints ensure that after the occurrence of c in P_5 every property is satisfied, which does mean that c can actually not occur due to a false precondition. This is the expected behaviour, because this part of the process allows data changes violating the desired safety property, and the timing properties over the processes guarantee that this process part is unreachable. The process P^t contains the timing properties of $ProcFree(HandleEM \parallel Running)$:

$$\begin{aligned}
P^t &\stackrel{c}{=} LTdetEM \bullet \wedge x' < 8 \wedge x' = len \wedge len > 0 \rightarrow P^t_1\\
P^t_1 &\stackrel{c}{=} a \bullet x' \leq 8 \wedge x' = x + len \wedge len > 0 \rightarrow P^t_3\\
P^t_3 &\stackrel{c}{=} b \bullet \wedge y' = len \wedge y' \leq 8 \wedge x' \leq 8 \wedge x' = x + len \wedge len > 0 \rightarrow P^t_5\\
P^t_5 &\stackrel{c}{=} c \bullet \wedge y' = y + len \wedge y \geq 8 \wedge x' \leq 8 \wedge x' = x + len \wedge len > 0 \rightarrow \texttt{Skip}
\end{aligned}$$

$$\text{(axiom)}$$
$$\frac{x_3 > y_3 \vdash x_3 > y_3}{y_3 \geq x_3, x_3 > y_3 \vdash \mathit{false}} \text{(relation)(negation left)}$$
$$\frac{y_3 \geq x_3, x_3 > y_3 \vdash \mathit{false}}{y_3 \geq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + y_3, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false}} \text{(relation)}$$
$$\frac{y_3 \geq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + y_3, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false}}{\begin{array}{c} y = y_3 + \mathit{len}_3, y_3 \geq 8, x \leq 8, x = x_3 + \mathit{len}_3, \mathit{len}_3 > 0, y_3 = \mathit{len}_2, \\ y_3 \leq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + \mathit{len}_2, \mathit{len}_2 > 0, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false} \end{array}} \text{(relation)}$$
$$\text{(next-step)} \frac{\cdots \vdash \mathit{false}}{y = \mathit{len}_2, y \leq 8, x \leq 8, x = \mathit{len}_0 + \mathit{len}_1 + \mathit{len}_2, \mathit{len}_2 > 0, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P^t_5]\mathit{false}}$$
$$\frac{y = \mathit{len}_2, y \leq 8, x \leq 8, x = \mathit{len}_0 + \mathit{len}_1 + \mathit{len}_2, \mathit{len}_2 > 0, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P^t_5]\mathit{false}}{\begin{array}{c} y = \mathit{len}_2, y \leq 8, x \leq 8, x = x_2 + \mathit{len}_2, \mathit{len}_2 > 0, x_2 \leq 8, \\ x_2 = \mathit{len}_0 + \mathit{len}_1, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P^t_5]\mathit{false} \end{array}} \text{(relation)}$$
$$\frac{\cdots \vdash [P^t_5]\mathit{false}}{x \leq 8, x = \mathit{len}_0 + \mathit{len}_1, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P^t_3]\mathit{false}} \text{(next-step)}$$
$$\frac{x \leq 8, x = \mathit{len}_0 + \mathit{len}_1, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P^t_3]\mathit{false}}{x \leq 8, x = x_1 + \mathit{len}_1, \mathit{len}_1 > 0, x_1 < 8, x_1 = \mathit{len}_0, \mathit{len}_0 > 0 \vdash [P^t_3]\mathit{false}} \text{(relation)}$$
$$\frac{x \leq 8, x = x_1 + \mathit{len}_1, \mathit{len}_1 > 0, x_1 < 8, x_1 = \mathit{len}_0, \mathit{len}_0 > 0 \vdash [P^t_3]\mathit{false}}{x < 8, x = \mathit{len}_1, \mathit{len}_1 > 0 \vdash [P^t_1]\mathit{false}} \text{(next-step)}$$
$$\frac{x < 8, x = \mathit{len}_1, \mathit{len}_1 > 0 \vdash [P^t_1]\mathit{false}}{\mathit{true} \vdash [P^t]\mathit{false}} \text{(next-step)}$$

Figure 8.13.: Proof tree for checking that the event c in P_5 cannot occur due to timing constraints

We prove that P^t, in which we isolated the process sequence leading to the sub-process P_5, satisfies the property

$$\mathit{true} \vdash [P^t]\mathit{false},$$

stating that every property is true after executing event c in P_5. The proof tree[2] for this property is given in Fig. 8.13.

Verifying Side-Conditions for Oracle Rules

In the proof tree presented in the previous section, the oracle rules (assumption$_\Box$) and (box assumption$_\delta$) are applied three times. In order to get a correct proof for the desired collision freedom, we have to additionally check the side-conditions of the rules. Hence, we have to verify that the DC assumptions on the unknown parts actually imply the properties that are needed in the proof tree. The rules are applied in the following contexts:

$$\mathit{Cons}, \psi \vdash [\mathtt{Proc}_{\backslash \varnothing, V_3} \bullet \mathit{Rec}]\Box \mathit{safe} \tag{T1}$$
$$\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathtt{Proc}_{\backslash A_0, \varnothing} \bullet \mathit{WaitEM}]\Box \mathit{safe} \tag{T2}$$
$$\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathtt{Proc}_{\backslash A_0, \varnothing} \bullet \mathit{WaitEM}]\mathit{LTcurPos} \geq \mathit{sf}, \tag{T3}$$

where the first two lines are resolved by application of (assumption$_\Box$) and the last line by application of (box assumption$_\delta$). Note that the DC assumptions $\mathit{Run}1$, $\mathit{Run}2$, and

[2] The rule (relation) abbreviates relational conversions in the theory of reals.

$$
\begin{aligned}
&\neg(\lceil SysLength \leq 0 \lor SysTrgSpdDst \leq 0 \lor SysStpDst \leq 0\rceil ^\frown true) \land \\
&\neg(\lceil RTdrvRes \leq 0 \land (RTbrkMd < BrkMdEmBrk \lor \\
&\quad LTcurPos < RTstndEB + SysLength \lor \\
&\quad RTstndEB < RTcurPos + SysStpDst)\rceil ^\frown true) \land \\
&\neg\Diamond(\lceil RTstndEB \geq RTcurPos + SysStpDst \land RTbrkMd \geq BrkMdEmBrk \\
&\qquad \land LTcurPos \geq RTstndEB + SysLength\rceil \\
&\qquad ^\frown true ^\frown \lceil RTstndEB \leq RTcurPos \lor LTcurPos < RTstndEB + SysLength\rceil) \land \\
&\Diamond(\lceil LTcurPos \leq RTcurPos + SysLength \land RTdrvRes \leq 0\rceil)
\end{aligned} \tag{T1}
$$

$$
F \land \Diamond(\lceil LTcurPos \leq RTcurPos + SysLength \land RTdrvRes \leq 0\rceil) \tag{T2}
$$

$$
F \land \Diamond(\lceil LTcurPos < RTcurPos + SysTrgSpdDst + SysStpDst + SysLength\rceil) \tag{T3}
$$

$$
\begin{aligned}
F = {} & \neg(\lceil SysLength \leq 0 \lor SysTrgSpdDst \leq 0 \lor SysStpDst \leq 0 \lor \neg Init \lor \\
&\quad LTcurPos < RTcurPos + SysLength + SysStpDst + SysTrgSpdDst\rceil \\
&\quad ^\frown true) \land \\
&\neg(\lceil Init\rceil ^\frown true ^\frown \\
&\quad \lceil LTcurPos < RTcurPos + SysLength + SysStpDst + SysTrgSpdDst\rceil ^\frown true)
\end{aligned}
$$

Figure 8.14.: Side-conditions to be checked for (T1) up to (T3) of page 212

AckTime are not resolved by these oracle proof rules but instead by the translation-based approach for parallel unknowns. Therefore, we do not need to consider them here.

We follow the approach of Sect. 4.5 to verify these properties with DC model checking [MFHR08]. To simplify matters, we show all three cases by checking instances of formula (4.19), which is always possible by Remark 4.5.2 on page 94:

$$
(\lceil\psi\rceil ^\frown \text{true}) \land F \land \bigwedge_{ev \in A} \boxminus ev \land (\Diamond\lceil\neg\varphi\rceil),
$$

where ψ is the initial constraint and φ the property, we want to show. F is the constraint on the unknown process, i.e., *WaitEM* and *Rec* in this case.

Thus, this results in three proof tasks for (T1) up to (T3), listed in Fig. 8.14. We need to verify the unsatisfiability of each of these formulae (we omit the event part $\bigwedge_{ev \in A} \boxminus ev$, because it is not relevant here).

All of these proof tasks were successfully identified as unsatisfiable by ARMC. To this end, the ARMC verification of Syspect (cf. Sect. 7.2.2) was used to automatically perform the verification process. The runtimes for the verification are not significant: ARMC checks task (T1) in 1.78 sec, task (T2) in 0.45 sec, and task (T3) in 0.32 sec

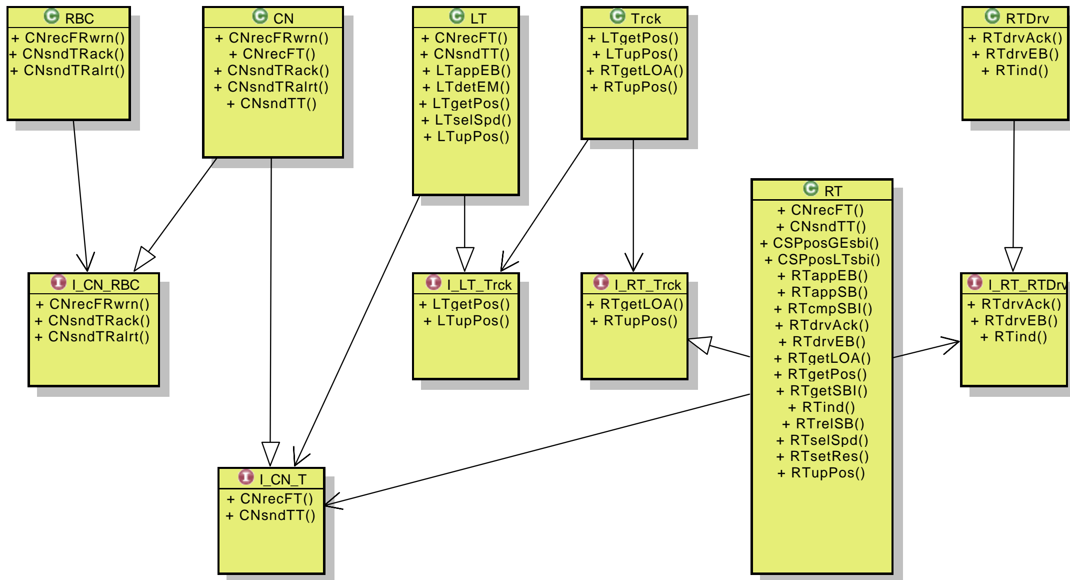


Figure 8.15.: Syspect class diagram for the ETCS case study

(see also Tab. 8.2). The runtimes for the translation process are in the same order of magnitude.

With this, the proof tree from the previous section is completed, and it is shown that the desired safety property is actually valid for the VA process.

8.2.5. Instantiating the VA

Having a VA that satisfies the desired safety property for the ETCS case study, the question is now whether the concrete ETCS model is an instance of this VA. To this end, we apply the Syspect tool (Sect. 7.1) to automatically check that this is actually the case.

Syspect Model of the ETCS Case Study

The original ETCS model as presented in Sect. 8.2.2 is specified as pure CSP-OZ-DC model. Ingo Brückner has remodelled the CSP-OZ-DC specification of the case study into a Syspect model in order to analyse the case study in [Brü08b] with the slicing plug-in of Syspect. It turned out that even though the specification can be reduced by slicing, it is still too large to be verified automatically [Brü08b]. We use this Syspect model of the case study (not being sliced) as a basis for checking the instantiation of the ETCS VA. Figure 8.15 presents a Syspect class diagram of the case study.

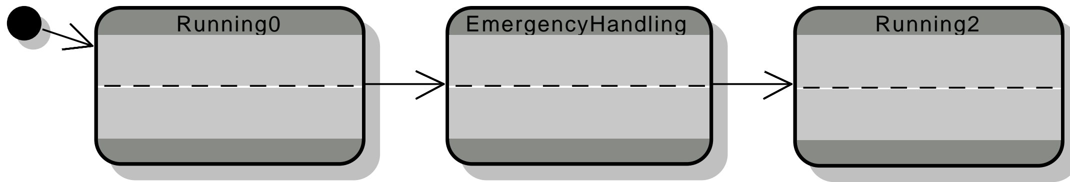


Figure 8.16.: Control structure of class *Trck* as modelled in Syspect

Changes to the Model

As Syspect does support only restricted CSP-OZ-DC specifications, some changes to the original ETCS specification became necessary. For instance, the **if** . . . **then** . . . **else**-construct in the CSP part has to be replaced by a standard CSP expression, and local variables occurring in the CSP part have to be replaced by corresponding variables of the OZ part. See [Brü08b] for details on these changes.

The instantiation checking approach of Syspect follows the line of Chap. 5. That is, the refinement check consists of two parts: a syntactic check of the structure of the model against the VA structure and a second check of the local assumptions. This makes some further modifications to the ETCS model necessary, because the structure of the Syspect model does not fit to the VA structure. In particular, the Syspect model consists of several parallel classes, partly with its own CSP part, while the VA is designed wrt. a single component. Hence, according to the discussion in Sect. 5.6, the VA structure must be somehow reflected by a single class. For this reason, we shift the control part of both train classes, *RT* and *LT*, into a protocol class. In this case, we choose the environment class *Trck* as protocol class, implementing the control structure of both trains. That is, instead of having a control structure for each train, the class *Trck* implements the structure of Fig. 8.16: the control structures are integrated into a single process with parallelism.[3] Figure 8.17 shows the class diagram of the extended model with new interfaces, which ensure by synchronisation that the train classes behave according to the protocol of *Trck*.

A part of the corresponding CSP process of *Trck* is pictured in Fig. 8.18. The CSP processes of the trains from the original case study are rearranged in this overall control structure. In doing so, the unknown process *ProcWaitEM* is instantiated with $Running_0$, *ProcAckTime* with $HandleEM_C$, *ProcRun* with $Running_1$, and *ProcRec* with $Running_2$. The sub-processes $RTRunning_i$ and $LTRunning_i$ are with small exceptions identical to the *Running* processes of *RearTrain* and *LeadTrain* in the original case study. Note that in Syspect no process references are possible. Thus, in the Syspect model identical sub-processes are realised by copying them.

[3] The current version of Syspect does not support parallel composition with synchronisation, which is important in this case study. Nevertheless, we use the interleaving operator of Syspect instead and take care manually that the synchronisation is correctly handled. (For instance, the LaTeX export is adapted.) This is possible, because when applying the VA approach, we do not need to compute the parallel composition of instances of unknown parts.

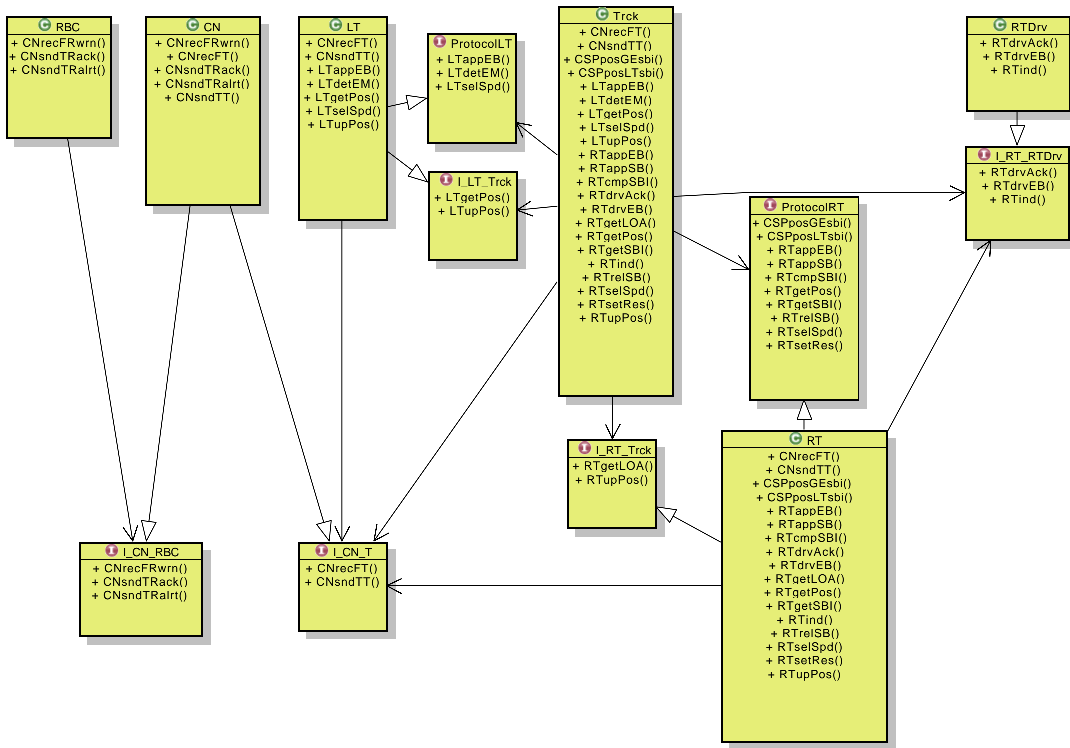


Figure 8.17.: Class diagram with *Trck* as protocol class

In order to simplify the verification, we made a further modification: in the original specification a phase can be changed at any time. For instance, the *LTdetEM* event can occur at any point in time during a train movement cycle. In the modified variant, we restrict the model such that the update cycle can only be left after a specific event, namely after an *RTupdPos* event. This is actually a simplification of the original model, because it could potentially cause an error if the cycle is left at an unexpected time. Nonetheless, the basic ideas of the original case study are preserved. The full CSP-OZ-DC specification generated from the Syspect model can be found in Appendix A.2.3.

8.2.6. Checking the Instantiation with Syspect

The VA plug-in of Syspect can be used to verify that this model is actually a correct refinement of the VA process ETCS-EM: the structural refinement check of Syspect returns immediately (< 1s) that the control structure of the model indeed matches the structure of the architecture.

To automatically verify that the local assumptions on unknowns are valid for instantiations of the unknown parts, we need to pay attention wrt. the parallel unknowns occurring in the VA process. In particular, the Syspect model comprises several com-

$$
\begin{aligned}
\texttt{main} &\overset{c}{=} \mathit{Running}_0 \mathbin{;} \mathit{Emergency}_C \mathbin{;} \mathit{Running}_2 \\
\mathit{Emergency}_C &\overset{c}{=} (\mathit{HandleEM}_C \underset{A}{\parallel} \mathit{Running}_C) \\
\mathit{Running}_C &\overset{c}{=} (\mathit{LTdetEM} \rightarrow \mathit{RTRunning}_1) \\
&\qquad \mathbin{;} ((\mathit{RTappEB} \rightarrow \texttt{Skip}) \mathbin{\Box} (\mathit{RTsetRes} \rightarrow \texttt{Skip})) \\
\mathit{Running}_0 &\overset{c}{=} \mathit{RTRunning}_0 \,|||\, \mathit{LTRunning}_0 \\
\mathit{Running}_1 &\overset{c}{=} \mathit{RTRunning}_1 \,|||\, \mathit{LTRunning}_1 \\
\mathit{Running}_2 &\overset{c}{=} \mathit{RTRunning}_2 \,|||\, \mathit{LTRunning}_2 \\
A &= \{\mathit{LTdetEM}, \mathit{RTappEB}, \mathit{RTsetRes}\}
\end{aligned}
$$

Figure 8.18.: Part of the CSP process representing the control structure of the ETCS model

ponents with a CSP part running in parallel: *Trck*, *CN*, *RTDrv*, and *RBC*, where the CSP part of *Trck* matches the architecture structure. In Sect. 5.6 we noticed that further parallel components cause no problems as long as the control flow of the components is restarted at the same time a process refining an unknown component is entered. We use the syntactic check of Syspect (cf. 7.3.3) to verify that this is actually the case for the ETCS model, as *CN*, *RTDrv*, and *RBC* are only synchronised with *HandleEM*. Hence, they can be ignored when checking the assumptions in all cases except for the verification of *AckTime* (the assumption on *HandleEM*). Syspect automatically ignores irrelevant components. Moreover, one can also manually deselect irrelevant components for each assumption check if desired, but for our experimental results we always used the components that are automatically selected by Syspect.

Moreover, we must show that the instantiating processes of the unknown parts cannot introduce new interferences. This is done by verifying that all system variables of the instantiation satisfy one of the three conditions in Thm. 6.1.11. As already mentioned, Syspect does not support parallel compositions with synchronisations; hence, this step has to be performed manually. To this end, the system variables that are accessed in $\mathit{HandleEM}_C$ and $\mathit{Running}_C$ have to be analysed wrt. the three conditions in Thm. 6.1.11. Table 8.1 lists the variables occurring in the processes. The column 'access' quotes how the variables are accessed by the parallel components. If the variables are marked with 'local' condition 1. of Thm. 6.1.11 is applicable, 'sync' indicates that condition 2. and 'shared' that condition 3. is applicable. The emergency and the service brakes are modelled by one variable *RTbrkMd*. But actually, accesses to the emergency brake are only performed synchronously, while the service brake is only accessed in $\mathit{Running}_C$. Hence, we split the variable into two parts in Tab. 8.1. According to Thm. 6.1.11 'local' and 'sync' variables are safe in that they do not introduce interferences. For the three variables marked with 'shared' we must show that the assumption on $\mathit{HandleEM}_C$, namely *AckTime*, is actually independent

from changes to these variables. This is done by a parallel composition with an additional component setting the shared variables to arbitrary values when checking the *AckTime* assumption.

Furthermore, one has to validate that the initial constraint used to prove the correctness of the VA, here

$$Cons \land Init \land LTcurPos \geq RTcurPos + SysLength + SysTrgSpdDst + SysStpDst, \tag{8.9}$$

is an initial constraint of the Syspect specification. We instantiate *Init* by

$$\begin{aligned} Init = {} & RTcurPos \leq SysRTStrtPos \land RTsbi \leq SysRTStrtSBI \\ & \land RTcurSpd \leq SysMaxSpd \land SysTrgSpd \leq LTcurSpd \\ & \land LTcurPos \geq SysLTStrtPos \land TrLTpos \leq SysLTStrtPos \\ & \land LTbrkMd \leq BrkMdNone. \end{aligned}$$

The Syspect specification (and the CSP-OZ-DC specification; cf. Appendix A.2.3) contains the Z definitions

$$\begin{array}{|l} SysLength, SysStpDst, SysTrgSpdDst, SysLTStrtPos, SysRTStrtPos, \\ SysRTStrtSBI, SysLTStrtPos, SysOffDst : Pos \\ SysMaxSpd, SysTrgSpd : Spd \\ \hline 0 < SysTrgSpd < SysMaxSpd \\ 0 < SysLength \\ 0 < SysStpDst < SysTrgSpdDst \\ SysRTStrtSBI = SysLTStrtPos - SysLength - SysTrgSpdDst - SysStpDst \\ \qquad\qquad\qquad\qquad - SysMaxSpd \\ SysRTStrtPos < SysRTStrtSBI \end{array}$$

and initial constraints

$$\begin{aligned} & RTcurPos = SysRTStrtPos \land SysTrgSpd < RTcurSpd \land RTcurSpd \leq SysMaxSpd \land \\ & RTsbi = SysRTStrtSBI \land LTcurPos = SysLTStrtPos \land SysTrgSpd \leq LTcurSpd \land \\ & LTcurSpd \leq SysMaxSpd \land LTbrkMd = BrkMdNone \land TrLTpos = SysLTStrtPos \end{aligned}$$

implying (8.9) as desired. We have also verified this property with Syspect (see Tab. 8.2).

Finally, as ARMC and SLAB support only linear arithmetical constraints over reals, we need to replace the symbolic time constants and depending constants, e.g., the braking distance, by concrete constants.[4] Also, as ARMC only supports real-valued variables we have to replace all expressions over Boolean or free types by corresponding real variables.

[4] We choose the constants as small as possible, because it turned out that the verification runtimes significantly increase with higher constants.

Table 8.1.: System variables of $HandleEM_C$ and $Running_C$

$HandleEM_C$		$Running_C$		
read	write	read	write	access
	RTbrkMd(EB)		RTbrkMd(EB)	sync
	RTdrvRes		RTdrvRes	sync
	RTstndEB		RTstndEB	sync
RTcurSpd			RTcurSpd	shared
RTsbi			RTsbi	shared
RTcurPos			RTcurPos	shared
	CNi			local
	CNm			local
	RTCSPHpos			local
	RTCSPHsbi			local
			RTbrkMd(SB)	local
			LTcurPos	local
			TrLTpos	local
			RTCSPRloa	local
			RTCSPRsbi	local
			TrRTpos	local
			TrLTpos	local
			LTbrkMd	local
			LTcurSpd	local

Table 8.2.: Experimental results for the ETCS case study

Task		Syspect	ARMC with slicing	SLAB	Syspect	ARMC without slicing	SLAB
Oracle rules	T1	-	-	-	0.5s	1.78s	11.27s
	T2	-	-	-	0.25s	0.45s	1.98s
	T3	-	-	-	0.12s	0.32s	1.01s
Assumptions	WaitEM	14s	7m 11s	t.o.	40s	11m 5s	t.o.
	AckTime	10s	1m 21s	5m 50s	10s	9h 55m	mem.
	Run1	6s	242m	t.o.	15s	t.o.	t.o.
	Run2	5s	1m 2s	t.o.	8s	1m 44s	t.o.
	Rec	7s	1m 57s	t.o.	27s	5m 56s	t.o.
Initial Constraint		2s	0.53s	2.60s	8s	22s	2m 18s
Structural refinement		<1s			<1s		

Results obtained on a standard desktop computer
(AMD Athlon Dual Core, 2500 MHz, 4 GB RAM).

With these preparations, the Syspect plug-in can be used to check the structural refinement relation between CSP-OZ-DC specification and VA process, to compute the sub-processes that shall guarantee the local assumptions, and to export the proof obligations into ARMC or SLAB syntax. By this means, we were able to show that all local assumptions are valid in the concrete specification. Table 8.2 summarises the results for all verification runs. The column ‘Syspect’ lists the runtimes for generating the input files for the model checkers, including the computation of the relevant sub-processes for each assumption, the runtimes for the translation of DC formulae into PEA, for computation of the parallel composition of the considered PEA, and for applying some heuristic simplifications and slicing. The columns ‘ARMC’ and ‘SLAB’ lists the runtimes for model checking the proof tasks. The left part of the table contains results with slicing enabled and the right part the results without slicing.

The listed runtimes show that the time for computing the model checker input and for verifying the side-conditions of the oracle rules is not of much consequence. The runtimes for actually model checking the proof tasks for the assumptions is in fact much higher—in the order of minutes or even hours for larger parts. The reason is that the unknown processes are instantiated with processes composed in parallel. For instance, in the case of the task *WaitEM* the model checked system part consists of 40 real-valued variables, more than 8500 transitions, and 179 (symbolic) locations. The long runtime of task Run1 originates from the dependence on time constants: the longer the distance the train can move in a specific time-interval—which is dependent on the time constants for the train movement cycle—the longer the runtime of the model checker for this task. Thus, we choose small time constants in $AckTime$, $Run1$, and in the Syspect model. In addition, we also applied the model checker SLAB on the proof tasks, but we got time-outs (in the table, ‘t.o.’ stands for time-out, ‘mem.’ for memory error) for some of the problems.

Furthermore, Table 8.2 demonstrates the benefits of slicing when used in combination with the VA decomposition as argued in Sect. 6.4.1. When slicing the entire specification with our desired safety property as slicing criterion, almost no advantage is achieved, because nearly no component is syntactically independent from the remainder of the specification. But with the VA approach only sub-components are to be verified, by what for each sub-tasks large parts of the specification are completely independent. Thus, slicing can be used to identify and remove the independent parts. The table shows that this is particularly the case for property *AckTime* that only depends on the timing parts of the specification.

8.2.7. Discussion

These results demonstrate that the VA approach allows us to handle problems that cannot be solved by direct verification. The VA approach ensures that the desired global safety property (8.7) is valid for the Syspect model of the ETCS case study. Although the case study is not realistic in all respects—e.g., the train braking is mod-

elled by approximation since CSP-OZ-DC does not allow us to model general hybrid systems—it demonstrates the VA approach in the context of a complex system incorporating communicating parallel components with data as well as real-time behaviour. On the other hand, without the VA approach, it was not possible to automatically verify the model (time out after over 80 hours) even with further decomposition approaches like slicing. Note that this holds likewise for the slightly simplified model from the previous section. Even with slicing enabled, the exported file has a size of over 1.5 GB and contains a hundred thousand of transitions.

Note that the original ETCS case study in [FM06, MFHR08] has not been developed with respect to a layered protocol design. Instead it turned out after designing the model that it behaves according to a protocol that can also be used to structure the verification of the model. In particular, with the graphical tool support given by Syspect, the VA approach provides a practical instrument to (in the first instance) manually analyse the structure of a protocol. Then automated methods can be used to verify that the origin model is actually an instantiation of the protocol.

When comparing the runtimes for checking the proof tasks to the results of the original case study in [MFHR08], one will notice that they are around the same order of magnitude. In the original case study, the proof tasks were manually adjusted, and for each task a dedicated set of relevant automata was chosen, whereas here, the relevant components were selected according to the VA structure. The proof tasks are not directly comparable, because in [MFHR08] they are mostly divided into further sub-tasks.

Once having a specification structured by a VA protocol, it can be reused to verify different concrete models instantiating the VA protocol. A further aspect is that when modifying a VA protocol, some of the proof obligations may remain valid without the need to verify them again. For instance, it is easy to extend the ETCS VA to an overall cyclic structure: in the recovery phase the system can be reset to a safe state, and afterwards, the VA process is started again. To this end, we just have to adapt the unknown process *ProcRec* such that it can terminate in a safe state. When modifying the concrete CSP-OZ-DC model in this way and trying to directly verify it with ARMC, the exported transition system becomes completely untreatable with over a million transitions and an input file size of several gigabytes. On the contrary, most of the proof tree for the VA remains the same, we only have to apply the rule for recursive behaviour, (box loop), and show that after termination of *ProcRec* the system is in a safe state again. To get a correct instantiation of the modified VA, one needs to change only the implementation of the *ProcRec* process and the protocol. Then it suffices to verify the structural refinement relation and the assumption of *ProcRec* to get a concrete model with the desired property.

9 Conclusions

> What was it, in the end? – What it always is. A handful of yarn; a little weaving and stitching; some embroidery perhaps. A few loose ends, but that's only to be expected...
>
> *(The Fates, in* The Kindly Ones, *Neil Gaiman)*

9.1. **Discussion** 224
9.2. **Alternative Approaches** 227
9.3. **Perspectives** 228

The main focus of this thesis has been on homogeneously integrating different techniques in a formal framework in order to enable exact analysis of complex real-time systems. To this end, a Verification Architecture (VA) approach has been introduced, which carries over the informal design pattern approach to formal verification.

VAs are defined in terms of a novel CSP dialect that combines CSP processes with data and additional temporal constraints over unknown parts in order to define reusable, behavioural protocols for large classes of systems.

Our new sequent-style proof calculus allows us to verify VAs by a combination of proof rule based reasoning and a suited verification technique for the temporal constraints. We have especially examined how parallel unknown components are verified in the calculus.

We have built on combined specification languages that are particularly suited to capture heterogenous system dimensions. As a proof of concept, we have considered instantiations of VAs by CSP-OZ-DC specifications and have given a syntactic proof rule to establish structural refinements. Instances of the VA, which are structural

refinements of the VA process and, additionally, satisfy the temporal constraints, inherit all safety properties of the VA.

VAs and CSP-OZ-DC models are specified with the graphical UML tool Syspect, by which the VA approach is also well-integrated with informal analysis techniques for complex systems. Using Syspect, we have successfully verified a CSP-OZ-DC-specified train control system taken from the AVACS sub-project R1 that is too complex to be verified without further decomposition techniques.

In this concluding chapter, we evaluate the VA approach and comment on alternative methods as well as possible future work.

9.1. Discussion

The key achievements of this thesis are

- a framework for specifying and verifying behavioural design patterns for combined specification languages,
- pattern-guided compositional verification of safety properties for concrete, complex specifications that instantiate a VA by checking structural refinement and verifying the local assumptions on unknown parts,
- the semantical analysis of VAs with respect to shared variable synchronisation and event-based communication of parallel components.

Language for Verification Architectures. To develop a language for the specification of VAs, we have to cope with the following discrepancy: On the one hand, we need a sufficient sophisticated formalism to treat the different system dimensions of complex real-time systems. On the other hand, the formalism has to be simple enough to be suited for formal analysis.

For this reason, we have developed the generic and light-weight CSP extension eCSP that is able to describe the same system dimensions as CSP-OZ-DC and a lot of other similar languages but with a very lean, CSP-based syntax. In addition, data constraints are integrated by many-sorted first-order formulae and temporal or real-time constraints by an arbitrary logic with the same semantical domain as eCSP. However, eCSP is not thought as replacement for existing combinations of formalisms that are usually designed with a different focus. For instance, CSP-OZ-DC combines well-investigated and standardised formalisms, and CSP-OZ-DC developers can make use of a large amount of syntactic constructs predefined in the Z standard. Instead, eCSP is intended for formal analysis of combined formalisms, and it can be used in combination with them, as it has been presented in this work.

In Sect. 6.3, we have discussed how far eCSP can be considered as complete in the sense that the language is strong enough to express all reasonable decompositions of combined specifications. We have shown that all CSP-OZ-DC specifications can

indeed be reformulated into an eCSP process, provided that the language on the unknown parts can express all necessary properties. Moreover, if we allow a small modification to the architecture process to capture global timing properties, then a stronger completeness result holds: all reasonable decompositions can be expressed with eCSP. That is, given a CSP-OZ-DC model that structurally refines an eCSP process, one can find assumptions on all unknown processes such that VA and model satisfy the same safety properties.

The eCSP formalism and the VA approach are not bound to CSP-OZ-DC, but can be used with other CSP-based combined formalisms, e.g., [WC01, SLD08], for which syntactic instantiation rules can be defined similarly to the rule from Sect. 5.1. However, this is obviously most convenient if the combined formalism is based on CSP. But it is to be assumed that the basic ideas of eCSP can be carried over to other process languages like CCS [Mil80] or the π-calculus [Mil99].

Reduction of complexity. VAs with unknown components decompose a global verification property into local proof obligations. By this, a concrete model can be compositionally verified by proving that the local proof obligations are valid. The underlying idea is that the complexity of the local proof tasks is reduced in comparison to the direct verification of the global property. In the case of the parallel composition of several components that refine parallel unknown processes of a VA, this is indeed the case, because the blow-up when computing the parallel composition of the parallel components is avoided; instead every component is checked on its own.

In the case of sequential unknown parts there is also a reduction in complexity, although the complexity of verifying sequential parts is only additive even when verifying the entire concrete model in a single step. Nevertheless, there is an advantage in verifying the local properties if a sequential component is repeated many times (then the local property is to be verified only once) or if it is part of a recursive process structure. In the latter case, the invariant of the recursion is already contained in the VA. Thus, it is not necessary to recompute the invariant for every instantiating and possibly very large model. Instead, the local proof obligations of sequential components contained in the recursion describe how the components contribute to the invariant for each cycle of the recursion. Moreover, as elucidated at the end of Chap. 8, when modifying a VA often large parts of a proof tree can be reused. For an instance of the VA, only the modified unknown parts and the structural refinement relation need to be rechecked.

The complexity can be further reduced when it is possible to apply additional reduction techniques to the components. An example is the slicing reduction technique (Sect. 6.4.1): if the global property depends on the whole concrete specification no slicing is possible; but if the global property can be inferred from local proof obligations of sequential components, slicing may be applicable for each of the sequential components. We have demonstrated this characteristic for the ETCS case study in Sect. 8.2.6. Generally speaking, the VA approach allows us to structure global verific-

ation tasks into sub-tasks of sub-components that are easier to verify due to arbitrary reasons dependent on the applied verification tool.

Moreover, the VA may contain large concrete parts (i.e., without unknowns), possibly with parallelism, that have to be verified only once—when checking the architecture. Thus, for these parts the complexity in verifying a concrete model is also reduced, because they do not need to be checked again for every instantiation. The verified pattern can be reused for arbitrary concrete models.

The complexity is, not least, also reduced in an engineering sense: structuring a verification problem according to an abstract protocol formulated in terms of a VA may help a system analyst to increase the understanding of a system or model.

Concurrency in Verification Architectures. In Sect. 6.1, we have discussed how to treat parallel unknown components with the proof calculus for eCSP. Two solutions have been provided. However, the given answers sacrifice some of the ideas of VAs to achieve the desired goal. The translation-based approaches for parallel unknowns modify the structure of the VA process; for the region construction it is exponentially blown-up. In particular, when having a timing property for the entire VA process, which is modelled by an unknown component running in parallel to the remainder of the VA, the entire structure of the VA is changed with the translation-based approaches. The approach we used for the ETCS case study expands the structure less than the region construction, but clutters up the process with timing constraints. The R-G approach for parallel unknowns allows for compositional reasoning in the sequent calculus, but the presented rules are only applicable if real-time properties are not relevant for the global property. Therefore, these results could probably be improved by developing a compositional real-time R-G rule, which can be integrated into the proof calculus similarly to the existing R-G rule.

However, it is no surprise that the verification of parallel processes with timing constraints and additional shared variables is difficult: it introduces a deep interdependence between the components due to mutual interference of variables. Hence, one may wonder if it was a good design decision to include shared variables along with the timing constraints. The reason for this decision was the desire for a highly flexible solution, allowing for unknown components with a large degree of freedom to capture a large class of concrete systems, involving time, shared variables access, and synchronisation over events. For example, when using VAs to decompose CSP-OZ-DC classes one indeed needs shared variable access, because within classes every operation may access the system variables belonging to the class. Nevertheless, it is no problem to restrict oneself to components without shared variable access. Then, parallel components communicate over synchronising events. Dedicated proof rules for such specific situations can be developed by adapting existing solutions for Assume-Guarantee reasoning, e.g., [MC81, Jon81, CLM89, AL93, AL95, FMS98, KV98]. So, the presented solution with general parallel components involving shared variables and event-based communication gives rise to a large flexibility; dedicated proof rules can be developed for specific situations where a smaller degree of freedom is desired.

9.2. Alternative Approaches

The core of the VA approach is the extended CSP dialect with the proof calculus for proving safety properties. Of course there are different ways to describe abstract protocols for combined specification languages. In the following, we elucidate on two alternative approaches for the specification of VAs and motivate why we have chosen to introduce eCSP instead.

Pure DC Approach

In [DHO06, DMO^{+}07] a behavioural pattern for cooperating traffic agents was introduced. Concrete models were formulated as hybrid automata, and the behavioural pattern was completely described in terms of DC formulae. The DC formulae were manually checked for correctness.

In a similar way, one could also formulate VAs for the use with combined specifications. If automatic analysis is desired, one needs to restrict usage of DC formulae to a sub-set of DC that can be handled with automated methods, e.g., DC counter-example traces [Hoe06]. In our experiments it turned out that the number of parallel DC formulae that can be handled with this approach is rather limited. Thus, it is not realistic to automatically analyse large protocol structures in this way. Another possibility is to use one of the proof-rule based verification approaches for DC that can be found in, e.g., [SS94, Hei99]—but even then one has to deal with a large amount of parallel DC formulae.

However, we preferred a more structured approach using CSP processes to describe the protocols with explicit unknown parts and generic temporal constraints. By this means, the control flow part of the system is described directly in CSP, and it is not necessary to encode the control flow with DC formulae. Moreover, the temporal constraints of the protocol are clearly separated by the protocol phases such that it suffices to analyse DC formulae for protocol phases that are simultaneously enabled. We showed in this thesis how a refinement relation between an eCSP protocol and a concrete specification is established; it is not clear how this can be achieved for a general set of DC formulae.

Pure Model Checking Approach

A second alternative approach is to straightforwardly verify VAs with existing model checking techniques for the combined specification language CSP-OZ-DC. That is, in order to apply the verification approach of [MFHR08], we have used parametric CSP-OZ-DC (with unknown components) to describe the protocol structure of a VA. DC formulae have been used to specify assumptions on specific phases of the protocol.

Since the assumptions cannot be integrated into the process structure of the CSP-OZ-DC specification (as long as the CSP-OZ-DC semantics is not to be changed), they are instead composed in parallel to the CSP-OZ-DC specification. For this reason,

the assumptions must contain the information for which phase of the protocol they are valid. The DC assumptions describe the behaviour of components under the assumption that the environment behaves correctly. So, given a CSP-OZ-DC model c, describing the protocol structure, and assumptions $F_1, \ldots, F_n$, a safety property G is proven correct by verifying that

$$c \models F_1 \wedge \ldots \wedge F_n \Rightarrow G.$$

Using the approach of [MFHR08], this formula can only be automatically verified if $F_1, \ldots, F_n$, and G are DC counterexample traces.

We successfully applied this approach to verify a simplified VA variant for the running example of Sect. 1.1. The protocol consists of the three phases $\mathit{FAR}, \mathit{REQ}$, and REC. The DC formulae for the assumptions are of the shape

$$\begin{aligned} \mathit{FAR} \equiv \neg(\mathit{true} ^\frown \updownarrow \mathit{grant} ^\frown \lceil \mathit{cr}_T > \mathit{AD} \rceil \wedge \boxminus \mathit{req} \\ ^\frown \not\updownarrow \mathit{req} ^\frown \boxminus \mathit{req} ^\frown \not\updownarrow \mathit{req} ^\frown \lceil \mathit{sf}_T \leq \mathit{DD} \rceil \wedge \boxminus \mathit{req} ^\frown \mathrm{true}). \end{aligned}$$

The safety property for the architecture is $\neg(\lceil \mathit{true} \rceil ^\frown \lceil \mathit{sf}_T \leq 0 \rceil)$. It has been verified in 62 seconds using ARMC by proving the unsatisfiability of the following property for the CSP-OZ-DC model:

$$\mathit{FAR} \wedge \mathit{REQ}_1 \wedge \mathit{REQ}_2 \wedge \mathit{REQ}_3 \wedge \mathit{REC} \wedge \mathit{INIT} \wedge (\lceil \mathit{true} \rceil ^\frown \lceil \mathit{sf}_t \leq 0 \rceil),$$

where $\mathit{REQ}_1, \mathit{REQ}_2, \mathit{REQ}_3, \mathit{REC}$ are formulated as DC assumptions similar to FAR, and INIT is an initial DC formula. Appendix A.1.4 lists the DC formulae and the simple CSP-OZ-DC specification for the architecture.

However, for slightly more complex protocols this approach fails because of the same reason as for the pure DC approach: the number of parallel assumptions—that are verified against a CSP-OZ-DC specification, which is also translated into several parallel automata—lead to a blow-up of the state space that cannot be handled with the model checking approach. Our VA approach overcomes this difficulty, because we usually do not have to consider all assumptions at the same time.

9.3. Perspectives

The achievements of this thesis can be extended in several directions.

Automatisation. The most important (practical) extension of this work is the development of tool support for the sequent-style calculus of Chap. 4, ideally with an integration into the Syspect tool. The proof trees for the examples of Chap. 8, which can be found in the appendix, show that even for small examples sequent-style proofs become unclear or even infeasible without tool-support. There are promising results of verification tools for similar calculi [HBB+05, BHS07, PQ08]. These tools are able

to solve a lot of intermediate proof steps fully automatically such that only in a few cases user interaction is necessary. It is likely, that the proof trees for our case studies can be solved nearly completely without user-interaction with appropriate tool support.

On the theoretical side, automated generation of assumptions for architectures is an interesting research question: Given a VA process with unknown parts and a concrete model, how can the weakest assumptions be generated such that the concrete model is an instance of a VA satisfying a desired safety property. Similar questions are examined in the AVACS sub-project S1; e.g., in [FSB06, FPS08, FPS10] assumptions (in terms of automata) are generated for given networks of timed automata. A similar question is whether assumptions of a given architecture can be weakened without violating desired properties. Finally, one may also ask how the sequential structure of a VA can be derived from a given concrete component. In Sect. 6.4.2 we have argued that the communication closed layer principle for timed systems may be applicable [OS10].

Proof rules. We have discovered in Thm. 4.4.2 that our proof calculus for VAs is incomplete due to the data in eCSP and the logic, but it is still an open question whether a relative completeness result can be established.

A further direction of possible extensions is to supplement the proof rules of this work. Additional proof rules for checking refinement can be developed that allow checking of larger classes of systems with a syntactical check. At the moment our refinement rules comprise only processes that nearly coincide except for the unknown parts. The rules could be extended to syntactic constructs that have a different shape but produce the same semantics (e.g., in trace semantics external choice and internal choice can be identified).

Additionally, in Sect. 9.1 we have already discussed the need for more sophisticated R-G rules with respect to timing dependencies between parallel components. The possible extension of the calculus to allow reasoning about timing properties with sequent-style proof rules has been explicitly analysed in Sect. 6.2.

Our formalisms and proof rules are based on the trace semantics of CSP. With the current approach it is therefore not possible to solve liveness and deadlock properties. It is of interest though, how far our results carry over to a more sophisticated semantics comprising a notion of deadlock or divergence. However, it is to be expected that there will be some barriers, because currently nothing forces an unknown component to progress eventually. Additional fairness properties on processes will be required. Also, with the refinement rules of Chap. 5, a refining model may prevent any progress, i.e., it can introduce deadlocks, which causes no problems as long as only safety properties are considered.

Applications. Up to now, we have examined only case studies from the transportation domain. Thus, it would be interesting to develop and validate further verification

patterns from a different application context. Likewise, the VA approach is still to be evaluated with respect to different formalisms for the assumptions and for the refining model.

Appendix

A Case Study Material

A.1. Train Control System of the Running Example

A.1.1. Proof Tree for the VA

In this section, the full proof tree to verify the correctness of the VA from Fig. 8.2 is given. The desired safety property is

$$sf > RD > 0 \vdash [System]\Box sf > 0.$$

We use the following abbreviations and definitions within the proof tree:

$$\begin{aligned}
System &\overset{c}{=} \overline{System} \,;\, System \\
\overline{System} &:= (FAR \,;\, check \rightarrow \\
&\quad (fail \rightarrow REC \,\Box\, pass \rightarrow \texttt{Skip}) \,\Box\, extend \rightarrow \texttt{Skip}) \\
P^0 &:= (fail \bullet \varphi^0_{fail} \rightarrow REC^0 \,\Box\, pass \bullet \varphi^0_{pass} \rightarrow \texttt{Skip}) \\
REC^0 &:= \texttt{Proc}^{\infty}_{\setminus A,C} \bullet F^0_{REC} \\
REC^1 &:= \texttt{Proc}^{\infty}_{\setminus A,C} \bullet F^1_{REC} \\
F^0_{REC} &:= \neg\Diamond(\lceil sf_0 > 0\rceil ^\frown \lceil sf_0 \leq 0\rceil) \\
F^1_{REC} &:= \neg\Diamond(\lceil sf_1 > 0\rceil ^\frown \lceil sf_1 \leq 0\rceil) \\
\varphi^0_{fail} &:= sf' = sf_0 \wedge \neg ok_0 \\
\varphi^1_{fail} &:= sf_1 = sf_0 \wedge \neg ok_0 \\
\varphi^0_{pass} &:= sf' = sf_0 \wedge ok_0 \\
\varphi_0 &:= sf_0 = sf \wedge sf \leq RD \wedge \neg ok_0 \\
&\quad \vee sf_0 = sf \wedge sf > RD \wedge ok_0
\end{aligned}$$

The formulae T_{FAR_1}, T_{FAR_2}, and T_{REC} are the DC proof obligations that are proven with ARMC. We use additional rules (equal), (relation), and (renaming) for the insertion of equalities, merging of relations in the theory of reals, and renamings to simplify the presentation of the proof tree. In addition, the tree is split into several sub-trees. Note that the proof direction is from bottom to top. That is, the desired property can be found at the bottom of the tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{ok_0 = \text{false} \vdash ok_0 = \text{false}\; sf_1 > 0}\text{(axiom)}}{ok_0 = \text{false}, \neg(ok_0 = \text{false}) \vdash sf_1 > 0}\text{(negation left)}}{ok_0 = \text{false}, ok_0 = \text{true} \vdash sf_1 > RD}\text{(equal)}}{sf_0 = sf, sf \leq RD, ok_0 = \text{false}, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD}\text{(weakening left)}$$

$$\dfrac{\dfrac{\dfrac{}{sf_1 > RD \vdash sf_1 > RD}\text{(axiom)}}{sf_0 = sf, sf > RD, sf_1 = sf_0 \vdash sf_1 > RD}\text{(equal)}}{sf_0 = sf, sf > RD, ok_0 = \text{true}, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD}\text{(weakening right)}$$

$$\dfrac{\dfrac{sf_0 = sf, sf \leq RD, ok_0 = \text{false}, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD \quad | \quad sf_0 = sf, sf > RD, ok_0 = \text{true}, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD}{(sf_0 = sf \wedge sf \leq RD \wedge ok_0 = \text{false}) \vee (sf_0 = sf \wedge sf > RD \wedge ok_0 = \text{true}), sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD}\text{(or left)(and left)}}{sf > 0, RD > 0, \varphi_0, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD}\text{(weakening left)}$$

$$\dfrac{sf > 0, RD > 0, \varphi_0, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD \qquad \dfrac{}{sf > 0, RD > 0, \varphi_0, sf_1 = sf_0, ok_0 = \text{true} \vdash RD > 0}\text{(axiom)}}{sf > 0, RD > 0, \varphi_0, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD > 0}\text{(and right)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash [\mathtt{Proc}^{\infty}_{\setminus A,C}] sf_1 > RD > 0}\text{(assumption axiom)}}{\vdash [REC^1] sf_1 > RD > 0}\text{(process call)}}{\vdash \varphi^1_{fail} \Rightarrow [REC^1] sf_1 > RD > 0}\text{(implication right)(weakening right)}}{\vdash [fail \bullet \varphi^0_{fail}][REC^0] sf_0 > RD > 0}\text{(box step}_\delta\text{)}}{\vdash [fail \bullet \varphi^0_{fail} \rightarrow REC^0] sf_0 > RD > 0}\text{(prefix}_\delta\text{)}$$

$$\dfrac{\vdash [fail \bullet \varphi^0_{fail} \rightarrow REC^0] sf_0 > RD > 0}{sf > 0, RD > 0, \varphi_0 \vdash [fail \bullet \varphi^0_{fail} \rightarrow REC^0] sf_0 > RD > 0}\text{(weakening right)}$$

$$\dfrac{\dfrac{sf > 0, RD > 0, \varphi_0, sf_1 = sf_0, ok_0 = \text{true} \vdash sf_1 > RD > 0}{sf > 0, RD > 0, \varphi_0 \vdash sf_1 = sf_0 \wedge ok_0 = \text{true} \Rightarrow sf_1 > RD > 0}\text{(implication right)(and left)}}{sf > 0, RD > 0, \varphi_0 \vdash [pass] sf_0 > RD > 0}\text{(box step}_\delta\text{)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{sf > 0, RD > 0, \varphi_0 \vdash [fail \bullet \varphi^0_{fail} \rightarrow REC^0] sf_0 > RD > 0 \qquad sf > 0, RD > 0, \varphi_0 \vdash [pass] sf_0 > RD > 0}{sf > 0, RD > 0, \varphi_0 \vdash [P^0] sf_0 > RD > 0}\text{(box choice)(and right)}}{sf > 0, RD > 0 \vdash \varphi_0 \Rightarrow [P^0] sf_0 > RD > 0}\text{(implication right)}}{sf > 0 \wedge RD > 0 \vdash [check][(fail \rightarrow \ldots \mathtt{Skip})] sf > RD > 0}\text{(and left)(box step}_\delta\text{)}}{sf > 0 \wedge RD > 0 \vdash [check \rightarrow \ldots] sf > RD > 0}\text{(sequence}_\delta\text{)}$$

$$\dfrac{sf > RD > 0, F_{FAR} \overset{T_{FAR_2}}{\vdash_{DC}} [\mathtt{Proc}_{\setminus A,C}](sf > 0 \wedge RD > 0) \qquad sf > 0 \wedge RD > 0 \vdash [check \rightarrow \ldots] sf > RD > 0}{sf > RD > 0 \vdash [FAR][check \rightarrow \ldots] sf > RD > 0}\text{(process call)(box assumption}_\delta\text{)}$$

(11)

$$\dfrac{\text{(sequence}_\delta)\ \dfrac{\overset{(11)}{sf > RD > 0 \vdash [FAR][check \rightarrow \ldots]sf > RD > 0}}{sf > RD > 0 \vdash [FAR \fatsemi \ldots \mathtt{Skip}]sf > RD > 0} \qquad \dfrac{\dfrac{\dfrac{\overset{\text{(axiom)}}{sf_0 > RD > 0 \vdash sf_0 > RD > 0}}{sf > RD > 0, sf_0 > sf \vdash sf_0 > RD > 0}\ \text{(relation)}}{sf > RD > 0 \vdash sf_0 > sf \Rightarrow sf_0 > RD > 0}\ \text{(implication right)}}{sf > RD > 0 \vdash [extend]sf > RD > 0}\ \text{(box step}_\delta)}{sf > RD > 0 \vdash [\overline{System}]sf > RD > 0}\ \text{(box choice)(and right)}$$

(2)

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{\text{(axiom)}}{sf > 0, ok = \text{true}, sf_0 = sf \vdash ok = \text{true}, [REC^0]\Box sf_0 > 0}}{sf > 0, ok = \text{true}, sf_0 = sf, \neg(ok = \text{true}) \vdash [REC^0]\Box sf_0 > 0}\ \text{(negation left)}}{sf > 0, ok = \text{true}, sf_0 = sf, ok = \text{false} \vdash [REC^0]\Box sf_0 > 0}\ \text{(equal)}}{sf > 0, ok = \text{true} \vdash (sf_0 = sf \wedge ok = \text{false}) \Rightarrow [REC^0]\Box sf_0 > 0}\ \text{(implication right)(and left)}}{sf > 0, ok = \text{true} \vdash [fail][REC]\Box sf > 0}\ \text{(box step}_\delta)$$

(10)

$$\dfrac{\dfrac{\text{(skip}_\Box)\ \dfrac{\overset{\text{(axiom)}}{sf > 0, ok = \text{true}, \vdash sf > 0}}{sf > 0, ok = \text{true} \vdash [\mathtt{Skip}]\Box sf > 0} \qquad \dfrac{\dfrac{\dfrac{\overset{\text{(axiom)}}{sf > 0, ok = \text{true}, sf_0 = sf \vdash ok = \text{true}, [\mathtt{Skip}]\Box sf_0 > 0}}{sf > 0, ok = \text{true}, sf_0 = sf, \neg(ok = \text{true}), \vdash [\mathtt{Skip}]\Box sf_0 > 0}\ \text{(negation left)}}{sf > 0, ok = \text{true}, sf_0 = sf, ok = \text{false} \vdash [\mathtt{Skip}]\Box sf_0 > 0}\ \text{(equal)}}{sf > 0, ok = \text{true} \vdash (sf_0 = sf \wedge ok = false) \Rightarrow [\mathtt{Skip}]\Box sf_0 > 0}\ \text{(implication right)(and left)}}{sf > 0, ok = \text{true} \vdash [fail]\Box sf > 0}\ \text{(box step)} \qquad \overset{(10)}{sf > 0, ok = \text{true} \vdash [fail][REC]\Box sf > 0}}{sf > 0, ok = \text{true} \vdash [fail \rightarrow REC]\Box sf > 0}\ \text{(prefix}_\Box)$$

(9)

$$
\dfrac{
\dfrac{\dfrac{}{sf > 0, ok = \text{true} \vdash sf > 0}\,(\text{axiom})}{sf > 0, ok = \text{true} \vdash [\texttt{Skip}]\Box sf > 0}\,(\text{skip}_\Box)
\qquad
\dfrac{\dfrac{\dfrac{}{sf_0 > 0, ok = \text{true} \vdash sf_0 > 0}\,(\text{axiom})}{sf > 0, ok = \text{true}, sf_0 = sf, ok = \text{true} \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{skip}_\Box)(\text{equal})}{sf > 0, ok = \text{true} \vdash (sf_0 = sf \wedge ok = \text{true}) \Rightarrow [\texttt{Skip}]\Box sf_0 > 0}\,(\text{implication right})(\text{and left})
}{
sf > 0, ok = \text{true} \vdash [pass]\Box sf > 0
}\,(\text{box step})(\text{and right})
$$

$$
\dfrac{\dfrac{\dfrac{\overset{(9)}{sf > 0, ok = \text{true} \vdash [(fail \rightarrow REC)]\Box sf > 0} \qquad sf > 0, ok = \text{true} \vdash [pass]\Box sf > 0}{sf > 0, ok = \text{true} \vdash [((fail \rightarrow \ldots \texttt{Skip})]\Box sf > 0}\,(\text{box choice})(\text{and right})}{sf_0 > 0, ok_0 = \text{true} \vdash [P^0]\Box sf_0 > 0}\,(\text{renaming})}{sf > 0, RD > 0, sf_0 = sf, sf > RD, ok_0 = \text{true} \vdash [P^0]\Box sf_0 > 0}\,(\text{equal})(\text{weakening left})
$$

(6)

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{T_{REC}}{sf_1 > 0, F^1_{REC} \vdash_{DC} [\texttt{Proc}^{\infty}_{\setminus A,C}]\Box sf_1 > 0}}{sf_1 > 0 \vdash [\texttt{Proc}^{\infty}_{\setminus A,C} \bullet F^1_{REC}]\Box sf_1 > 0}\,(\text{assumption}_\Box)}{sf_1 > 0 \vdash [REC^1]\Box sf_1 > 0}\,(\text{process call})}{sf > 0, sf_0 = sf, sf_1 = sf_0, ok_0 = false \vdash [REC \bullet F^1_{REC}]\Box sf_1 > 0}\,(\text{weakening left})(\text{equal})}{sf > 0, sf_0 = sf \vdash (sf_1 = sf_0 \wedge ok_0 = false) \Rightarrow [REC \bullet F^1_{REC}]\Box sf_1 > 0}\,(\text{implication right})(\text{and left})}{sf > 0, sf_0 = sf \vdash [fail \bullet \varphi^0_{fail}][REC]\Box sf_0 > 0}\,(\text{box step}_\delta)
$$

(8)

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{}{sf_0 > 0 \vdash sf_0 > 0}\,(\text{axiom})}{sf > 0, sf_0 = sf \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{skip}_\Box)(\text{equal})
\qquad
\dfrac{\dfrac{\dfrac{}{sf_1 > 0 \vdash sf_1 > 0}\,(\text{axiom})}{sf > 0, sf_0 = sf, sf_1 = sf_0 \vdash [\texttt{Skip}]\Box sf_1 > 0}\,(\text{skip}_\Box)(\text{equal})}{sf > 0, sf_0 = sf \vdash (sf_1 = sf_0 \wedge ok_0 = \text{false}) \Rightarrow [\texttt{Skip}]\Box sf_1 > 0}\,(\text{implication right})(\text{and left})(\text{weakening left})
}{
sf > 0, sf_0 = sf \vdash [fail \bullet \varphi^0_{fail}]\Box sf_0 > 0
}\,(\text{box step})
\qquad
\overset{(8)}{sf > 0, sf_0 = sf \vdash [fail \bullet \varphi^0_{fail}][REC]\Box sf_0 > 0}
}{
sf > 0, RD > 0, sf_0 = sf, sf \leq RD \vdash [fail \bullet \varphi^0_{fail} \rightarrow REC]\Box sf_0 > 0
}\,(\text{prefix}_\Box)(\text{weakening left})
$$

(7)

$$\dfrac{\dfrac{(\text{axiom})}{sf_0 > 0 \vdash sf_0 > 0}}{sf > 0, sf_0 = sf \vdash [\texttt{Skip}]\Box sf_0 > 0}\ (\text{skip}_\Box)(\text{equal})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{axiom})}{ok_0 = \text{false}, \vdash ok_0 = \text{false}, [\texttt{Skip}]\Box sf_1 > 0}}{ok_0 = \text{false}, \neg ok_0 = \text{false} \vdash [\texttt{Skip}]\Box sf_1 > 0}\ (\text{negation left})}{ok_0 = \text{false}, ok_0 = \text{true} \vdash [\texttt{Skip}]\Box sf_1 > 0}\ (\text{equal})}{ok_0 = \text{false} \vdash (sf_1 = sf_0 \wedge ok_0 = \text{true}) \Rightarrow [\texttt{Skip}]\Box sf_1 > 0}\ (\text{implication right})(\text{and left})(\text{weakening left})$$

(box step)(weakening left)

$$sf > 0, RD > 0, sf_0 = sf, sf \leq RD \vdash [fail \bullet \varphi^0_{fail} \to REC]\Box sf_0 > 0 \quad (7)$$

$$ok_0 = \text{false}, sf > 0, sf_0 = sf \vdash [pass \bullet \varphi^0_{pass}]\Box sf_0 > 0$$

(box choice)(and right)(weakening right)

$$sf > 0, RD > 0, sf_0 = sf, sf \leq RD, ok_0 = \text{false} \vdash [P^0]\Box sf_0 > 0$$

$$sf > 0, RD > 0, sf_0 = sf, sf > RD, ok_0 = \text{true} \vdash [P^0]\Box sf_0 > 0 \quad (6)$$

(or left)(and left)

$$\dfrac{\dfrac{sf > 0 \wedge RD > 0, \varphi_0 \vdash [P^0]\Box sf_0 > 0}{sf > 0 \wedge RD > 0 \vdash \varphi_0 \Rightarrow [P^0]\Box sf_0 > 0}\ (\text{implication right})}{sf > 0 \wedge RD > 0 \vdash [check][(fail \to \cdots \to \texttt{Skip})]\Box sf > 0}\ (\text{box step}_\delta)$$

(5)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{sf_0 > 0, sf_0 = sf \vdash sf_0 > 0}\,(\text{axiom})}{sf > 0, sf_0 = sf \vdash sf_0 > 0}\,(\text{equal})}{sf > 0, sf_0 = sf \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{skip}_\Box) \qquad \dfrac{\dfrac{\dfrac{}{sf_0 > 0, sf_0 = sf \vdash sf_0 > 0}\,(\text{axiom})}{sf > 0, sf_0 = sf \vdash sf_0 > 0}\,(\text{equal})}{sf > 0, sf_0 = sf \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{skip}_\Box)}{sf > 0 \wedge RD > 0, \varphi_0 \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{or left})(\text{and left})(\text{weakening left})$$

$$\dfrac{sf > 0 \wedge RD > 0, \varphi_0 \vdash [\texttt{Skip}]\Box sf_0 > 0}{sf > 0 \wedge RD > 0 \vdash \varphi_0 \Rightarrow [\texttt{Skip}]\Box sf_0 > 0}\,(\text{implication right})$$

$$\dfrac{\dfrac{\dfrac{}{sf > 0, RD > 0 \vdash sf > 0}\,(\text{axiom})}{sf > 0 \wedge RD > 0 \vdash [\texttt{Skip}]\Box sf > 0}\,(\text{skip}_\Box)(\text{and left}) \qquad sf > 0 \wedge RD > 0 \vdash \varphi_0 \Rightarrow [\texttt{Skip}]\Box sf_0 > 0}{sf > 0 \wedge RD > 0 \vdash [check]\Box sf > 0}\,(\text{box step})$$

$$\dfrac{sf > 0 \wedge RD > 0 \vdash [check]\Box sf > 0 \qquad \overset{(5)}{sf > 0 \wedge RD > 0 \vdash [check][(fail \rightarrow \cdots \rightarrow \texttt{Skip})]\Box sf > 0}}{sf > 0 \wedge RD > 0 \vdash [check \rightarrow \cdots \rightarrow \texttt{Skip}]\Box sf > 0}\,(\text{sequence}_\Box)(\text{and right})$$

$$\dfrac{\overset{(T_{FAR_2})}{sf > RD > 0, F_{FAR} \vdash_{DC} [\texttt{Proc}_{\setminus A,C}]sf > 0 \wedge RD > 0} \qquad sf > 0 \wedge RD > 0 \vdash [check \rightarrow \cdots \rightarrow \texttt{Skip}]\Box sf > 0}{sf > RD > 0 \vdash [FAR][check \rightarrow \cdots \rightarrow \texttt{Skip}]\Box sf > 0}\,(\text{box assumption}_\delta)$$

$$\dfrac{\dfrac{\overset{(T_{FAR_1})}{sf > RD > 0, F_{FAR} \vdash_{DC} [\texttt{Proc}_{\setminus A,C}]\Box sf > 0}}{sf > RD > 0 \vdash [FAR]\Box sf > 0}\,\begin{matrix}(\text{process call})\\(\text{assumption}_\Box)\end{matrix} \qquad sf > RD > 0 \vdash [FAR][check \rightarrow \cdots \rightarrow \texttt{Skip}]\Box sf > 0}{sf > RD > 0 \vdash [FAR \fatsemi \cdots \rightarrow \texttt{Skip}]\Box sf > 0}\,(\text{sequence}_\Box)(\text{and right})$$

(3)

$$\dfrac{\dfrac{\dfrac{}{sf > RD > 0 \vdash sf > 0}\,(\text{axiom})}{sf > RD > 0 \vdash [\texttt{Skip}]\Box sf > 0}\,(\text{skip}_\Box) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{sf_0 > 0 \vdash sf_0 > 0}\,(\text{axiom})}{sf_0 > 0 \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{skip}_\Box)}{sf > RD > 0, sf_0 > sf \vdash [\texttt{Skip}]\Box sf_0 > 0}\,(\text{relation})(\text{weakening left})}{sf > RD > 0 \vdash sf_0 > sf \Rightarrow [\texttt{Skip}]\Box sf_0 > 0}\,(\text{implication right})}{sf > RD > 0 \vdash [extend]\Box sf > 0}\,(\text{box step})$$

(4)

$$\dfrac{\dfrac{\overset{(3)}{sf > RD > 0 \vdash [FAR \fatsemi \cdots \to \texttt{Skip}]\Box sf > 0} \qquad \overset{(4)}{sf > RD > 0 \vdash [extend]\Box sf > 0}}{sf > RD > 0 \vdash [FAR \fatsemi \cdots \to \texttt{Skip}]\Box sf > 0 \land [extend \to \texttt{Skip}]\Box sf > 0}\text{(and right)}}{sf > RD > 0 \vdash [\overline{System}]\Box sf > 0}\text{(box choice)}$$

(1)

$$\dfrac{\dfrac{\dfrac{\overset{(axiom)}{sf > RD > 0 \vdash sf > RD > 0} \qquad \overset{(1)}{sf > RD > 0 \vdash [\overline{System}]\Box sf > 0} \qquad \overset{(2)}{sf > RD > 0 \vdash [\overline{System}]sf > RD > 0}}{sf > RD > 0 \vdash [\overline{System} \fatsemi System]\Box sf > 0}\text{(box loop gen)}}{sf > RD > 0 \vdash [FAR \fatsemi \ldots \Box\ extend \to System]\Box sf > 0}\text{(process equivalence)}}{sf > RD > 0 \vdash [System]\Box sf > 0}\text{(process call)}$$

A.1.2. CSP-OZ-DC Model of the Train Control System

The CSP-OZ-DC specification contains the Train and the RBC component running in parallel. The *sf* system variable from VA process (see Fig. 8.2) is instantiated with $ma - pos$ and RD is instantiated with $maxbd + maxcd$. The time parameter CT is also instantiated with a concrete value, because our tool chain does not support parametric time constants.

$maxcd, maxspd, maxbd : \mathbb{R}$

$maxcd = 110$
$maxspd = 10$
$maxbd > 0$

Train

```
method ack, check, fail, pass, fb
method sendCurPos : [curPos! : ℝ]
method sendMARequest : [curPos! : ℝ; reqDist! : ℝ]
chan extend : [newMA? : ℝ]
local_chan ADReached, applyEB
local_chan checkAD
local_chan notADReached
local_chan reqFailed, updPos, updSpd
```

$$
\begin{array}{rcl}
FAR & \stackrel{c}{=} & ((InitialState0 \\
 & ||| & InitialState1 \\
 & ||| & InitialState2) \,{}_9^\circ\, (check \rightarrow checked)) \\
InitialState0 & \stackrel{c}{=} & (updSpd \rightarrow State0) \\
InitialState1 & \stackrel{c}{=} & ((sendCurPos \rightarrow InitialState1) \\
 & \sqcap & \mathtt{Skip}) \\
InitialState2 & \stackrel{c}{=} & ((checkAD \rightarrow State5) \\
 & \sqcap & \mathtt{Skip}) \\
InitialState3 & \stackrel{c}{=} & (applyEB \rightarrow State8) \\
REC & \stackrel{c}{=} & (InitialState3 \,{}_9^\circ\, \mathtt{Stop}) \\
State0 & \stackrel{c}{=} & (updPos \rightarrow State1) \\
State1 & \stackrel{c}{=} & (InitialState0 \\
 & \sqcap & \mathtt{Skip}) \\
State3 & \stackrel{c}{=} & ((ack \rightarrow \mathtt{Skip}) \\
 & \sqcap & (reqFailed \rightarrow \mathtt{Skip}) \\
 & \sqcap & request) \\
State5 & \stackrel{c}{=} & ((ADReached \rightarrow request) \\
 & \Box & (notADReached \rightarrow InitialState2)) \\
State8 & \stackrel{c}{=} & (updSpd \rightarrow State9) \\
State9 & \stackrel{c}{=} & (updPos \rightarrow State8) \\
checked & \stackrel{c}{=} & ((fail \rightarrow REC) \\
 & \Box & (pass \rightarrow \mathtt{main})) \\
\mathtt{main} & \stackrel{c}{=} & ((extend \rightarrow \mathtt{main}) \\
 & \Box & (fb \rightarrow FAR)) \\
request & \stackrel{c}{=} & (sendMARequest \rightarrow State3)
\end{array}
$$

$pos, spd : \mathbb{R}$
$ma, ad, ok : \mathbb{R}$
$maxbd, bd : \mathbb{R}$
$maxcd, maxspd : \mathbb{R}$
$ebApplied, adReached : \mathbb{R}$

Init

$$ad > maxcd$$
$$ma - pos > maxbd + maxcd$$

enable_$sendCurPos$

$$curPos! : \mathbb{R}$$

effect_$sendCurPos$

$$curPos! : \mathbb{R}$$

$$curPos! = pos$$

effect_$check$

$$\Delta(ok)$$

$$(ma - pos \leq maxbd + maxcd \wedge ok' = 0)$$
$$\vee\ (ma - pos > maxbd + maxcd \wedge ok' = 1)$$

enable_$extend$

$$newMA? : \mathbb{R}$$

effect_$extend$

$$\Delta(ma)$$
$$newMA? : \mathbb{R}$$

$$ma' = newMA?$$
$$ma' - pos' > ma - pos$$

effect_$updPos$

$$\Delta(pos)$$

$$pos' = pos + spd$$

enable_$notADReached$

$$adReached \leq 0$$

enable_$pass$

$$ok \geq 1$$

effect_$checkAD$

$$\Delta(adReached)$$

$$(ma - pos \leq maxbd + ad \wedge adReached' = 1)$$
$$\vee\ (ma - pos > maxbd + ad \wedge adReached' = 0)$$

enable_$fail$

$$ok \leq 0$$

$\texttt{effect_}\mathit{updSpd}$
$\Delta(spd, bd)$

$0 \leq spd' \leq maxspd$
$0 \leq bd' \leq maxbd$
$ebApplied \geq 1 \Rightarrow ma - pos - spd' > bd'$
$ebApplied \geq 1 \Rightarrow spd' = 0 \vee spd' < spd$
$ebApplied \geq 1 \Rightarrow bd' = 0 \vee bd' < bd$

$\texttt{effect_}\mathit{applyEB}$
$\Delta(ebApplied)$

$ebApplied' = 1$

$\texttt{enable_}\mathit{ADReached}$

$adReached \geq 1$

$\texttt{enable_}\mathit{sendMARequest}$
$curPos! : \mathbb{R};\ reqDist! : \mathbb{R}$

$\texttt{effect_}\mathit{sendMARequest}$
$curPos! : \mathbb{R};\ reqDist! : \mathbb{R}$

$curPos! = pos$
$reqDist! = ad + maxcd$

$\neg\Diamond(\updownarrow updPos ^\frown (\lceil true \rceil \wedge (\ell < 1)) ^\frown \updownarrow updPos)$
$\neg((\lceil true \rceil \wedge \boxminus fail) ^\frown \updownarrow fail ^\frown (\lceil true \rceil \wedge \boxminus fail \wedge \boxminus check \wedge \ell > 10) ^\frown true)$

RBC

$\texttt{method}\ extend : [newMA! : \mathbb{R}]$
$\texttt{method}\ getPosOfTrains : [otherTrainPos? : \mathbb{R}]$
$\texttt{chan}\ ack, fb$
$\texttt{chan}\ sendCurPos : [curPos? : \mathbb{R}]$
$\texttt{chan}\ sendMARequest : [curPos? : \mathbb{R};\ reqDist? : \mathbb{R}]$
$\texttt{local_chan}\ calcMA$
$\texttt{local_chan}\ cancelRequest$

$$
\begin{aligned}
InitialState5 &\overset{c}{=} ((getPosOfTrains \rightarrow InitialState5) \\
&\quad \Box\ (sendCurPos \rightarrow InitialState5)) \\
InitialState6 &\overset{c}{=} (sendMARequest \rightarrow State12) \\
State12 &\overset{c}{=} ((ack \rightarrow State13) \\
&\quad \sqcap\ InitialState6) \\
State13 &\overset{c}{=} ((calcMA \rightarrow State14) \\
&\quad \Box\ (cancelRequest \rightarrow InitialState6)) \\
State14 &\overset{c}{=} (extend \rightarrow InitialState6) \\
main &\overset{c}{=} ((InitialState5 \\
&\quad |||\ InitialState6) \fatsemi (fb \rightarrow \texttt{main}))
\end{aligned}
$$

$oTrainPos : \mathbb{R}$
$iMA : \mathbb{R}$
$minDist : \mathbb{Z}$
$iTrainPos : \mathbb{R}$

enable_*sendCurPos*

$curPos? : \mathbb{R}$

effect_*sendCurPos*

$\Delta(iTrainPos)$
$curPos? : \mathbb{R}$

$iTrainPos' = curPos?$

enable_*getPosOfTrains*

$otherTrainPos? : \mathbb{R}$

$otherTrainPos? > iMA$

effect_*getPosOfTrains*

$\Delta(oTrainPos)$
$otherTrainPos? : \mathbb{R}$

$oTrainPos' = otherTrainPos?$

enable_*cancelRequest*

$iMA \geq oTrainPos$
$\lor\ iTrainPos + minDist \geq oTrainPos$

enable_*extend*

$newMA! : \mathbb{R}$

effect_*extend*

$newMA! : \mathbb{R}$

$newMA! = iMA$

effect_*calcMA*

$\Delta(iMA)$

$iMA < iMA' < oTrainPos$
$iTrainPos + minDist < iMA'$

enable_*sendMARequest*

$curPos? : \mathbb{R};\ reqDist? : \mathbb{R}$

effect_*sendMARequest*

$\Delta(iTrainPos, minDist)$
$curPos? : \mathbb{R};\ reqDist? : \mathbb{R}$

$iTrainPos' = curPos?$
$minDist' = reqDist?$

A.1.3. CSP-OZ-DC Representation of the VA

The following specification is the CSP-OZ-DC representation of the Train Control VA, generated by Syspect (cf. Sect. 7.3.2).

Train

$\texttt{local_chan}\ check$
$\texttt{local_chan}\ extend$
$\texttt{local_chan}\ fail$
$\texttt{local_chan}\ fb$
$\texttt{local_chan}\ pass$

$$
\begin{array}{rcl}
FAR & \stackrel{c}{=} & (\texttt{Proc}_{\setminus A_1,\varnothing} \bullet FAR1 \wedge FAR2 \mathbin{;} (check \rightarrow checked)) \\
REC & \stackrel{c}{=} & (\texttt{Proc}^{\infty}_{\setminus A_2,\varnothing} \bullet REC \mathbin{;} \texttt{Stop}) \\
checked & \stackrel{c}{=} & ((fail \rightarrow REC) \\
 & \Box & (pass \rightarrow \texttt{main})) \\
\texttt{main} & \stackrel{c}{=} & ((extend \rightarrow \texttt{main}) \\
 & \Box & (fb \rightarrow FAR)) \\
A_1 & \stackrel{c}{=} & \{pass, check, fail, extend\} \\
FAR1 & \stackrel{c}{=} & \neg(true ^\frown \lceil ma > maxbd + maxcd + pos \rceil ^\frown (\ell < 10) ^\frown \lceil ma \leq pos \rceil ^\frown true) \\
FAR2 & \stackrel{c}{=} & \neg(true ^\frown (\lceil true \rceil \wedge (\ell > 10)) ^\frown true) \\
A_2 & \stackrel{c}{=} & \{pass, check, fail, extend\} \\
REC & \stackrel{c}{=} & \neg(true ^\frown \lceil ma > pos \rceil ^\frown \lceil ma \leq pos \rceil ^\frown true)
\end{array}
$$

$\texttt{enable_}fail$

$ok \leq 0$

$\texttt{effect_}check$

$(ma - pos \leq maxbd + maxcd \wedge ok' = 0) \vee (ma - pos > maxbd + maxcd \wedge ok' = 1)$

$\texttt{effect_}extend$

$ma' - pos' > ma - pos$

$\texttt{enable_}pass$

$ok \geq 1$

A.1.4. Alternative Architecture without eCSP

Protocol

The following parametric CSP-OZ-DC classes represent a Verification Architecture protocol without constraints on the unknown processes according to the model checking approach discussed in Sect. 9.2 for a simplified version of the running example.

$DD, AD : \mathbb{A}$

$0 \leq DD < AD$

RBC

method $request : [p : \mathbb{A}]$
chan $grant$
chan $deny$

$A == \{req, grant, deny\}$

$$\begin{aligned}
\texttt{main} &\overset{c}{=} P_{FAR} \mathbin{⨾} req \to P_{REQ} \\
&\quad \mathbin{⨾} (grant \to \texttt{main} \;\Box\; deny \to P_{REC}) \\
P_{FAR} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing} \\
P_{REQ} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing} \\
P_{REC} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing}
\end{aligned}$$

$sf_R : \mathbb{A}$

enable_*grant*

$sf_R > AD$

effect_*request*

$p? : \mathbb{A}$

$sf'_R = p?$

Train

method $deny$
method $grant$
chan $request : [p : \mathbb{A}]$

$A == \{req, grant, deny\}$

$$\begin{aligned}
\texttt{main} &\overset{c}{=} P_{FAR} \mathbin{⨾} req \to P_{REQ} \\
&\quad \mathbin{⨾} (grant \to \texttt{main} \;\Box\; deny \to P_{REC}) \\
P_{FAR} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing} \\
P_{REQ} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing} \\
P_{REC} &\overset{c}{=} \texttt{Proc}_{\backslash A,\varnothing}
\end{aligned}$$

$sf_T : \mathbb{A}$

Init

$sf_T > AD$

effect_*request*

$p! : \mathbb{A}$

$p! = sf_T$

Assumptions on the Protocol

- *The system is in the FAR phase.*

$$\begin{aligned} FAR \equiv \neg(true {}^\frown \updownarrow grant {}^\frown \lceil sf_T > AD \rceil \wedge \boxminus req \\ {}^\frown \not\updownarrow req {}^\frown \boxminus req {}^\frown \not\updownarrow req {}^\frown \lceil sf_T \leq DD \rceil \wedge \boxminus req {}^\frown \mathrm{true}) \end{aligned}$$

- *The system is in the REQ phase.*

$$\begin{aligned} REQ_1 \equiv \neg(true {}^\frown \updownarrow req {}^\frown \lceil DD < sf_R \leq AD \rceil & \\ \wedge \boxminus grant \wedge \boxminus deny \wedge \boxminus req & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny \wedge \not\updownarrow req & \\ {}^\frown \boxminus grant \wedge \boxminus deny \wedge \boxminus req & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny \wedge \not\updownarrow req & \\ {}^\frown \lceil sf_R \leq DD \vee AD < sf_R \rceil & \\ \wedge \boxminus grant \wedge \boxminus deny \wedge \boxminus req {}^\frown \mathrm{true}) & \end{aligned}$$

$$\begin{aligned} REQ_2 \equiv \neg(true {}^\frown \updownarrow req {}^\frown \lceil sf_T > DD \rceil & \\ \wedge \boxminus grant \wedge \boxminus deny & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny & \\ {}^\frown \boxminus grant \wedge \boxminus deny & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny & \\ {}^\frown \lceil sf_T \leq DD \rceil \wedge \boxminus grant \wedge \boxminus deny {}^\frown \mathrm{true}) & \end{aligned}$$

$$\begin{aligned} REQ_3 \equiv \neg(true {}^\frown \updownarrow req {}^\frown \lceil sf_T > AD \rceil & \\ \wedge \boxminus grant \wedge \boxminus deny & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny & \\ {}^\frown \boxminus grant \wedge \boxminus deny & \\ {}^\frown \not\updownarrow grant \wedge \not\updownarrow deny & \\ {}^\frown \lceil sf_T \leq AD \rceil \wedge \boxminus grant \wedge \boxminus deny {}^\frown \mathrm{true}) & \end{aligned}$$

- *The system is in the REC phase.*

$$REC \equiv \neg(true {}^\frown \updownarrow deny {}^\frown \lceil sf_T > DD \rceil {}^\frown \lceil \mathrm{true} \rceil {}^\frown \lceil sf_T \leq 0 \rceil {}^\frown \mathrm{true})$$

- *The system is initially in the INIT phase.*

$$INIT \equiv \neg(\boxminus req {}^\frown \not\updownarrow req {}^\frown \lceil sf_T \leq DD \rceil \wedge \boxminus req {}^\frown \mathrm{true})$$

A.2. ETCS Emergency Message Case Study

A.2.1. Original CSP-OZ-DC Specification from [MFHR08]

RearTrain(*ID* : *TrainID*; *StartPosition*, *StartSBI* : *Position*)

```
chan send : [m! : Message, id : {ID}]                      local_chan computeSBI : [loa?,sbi! : Position]
chan receive : [m? : Message, id : {ID}]                   local_chan applySB, releaseSB
chan updatePosition : [id : {ID},pos! : Position]          local_chan getPosition : [pos! : Position]
chan indication                                            local_chan getSBI : [sbi! : Position]
chan getLOA : [id : {ID}, loa? : Position]                 local_chan selectSpeed
chan driverAck, driverEB : [id : {ID}]                     local_chan applyEB
```

$$\begin{array}{rcl}
\mathtt{main} & \overset{c}{=} & Running \;|||\; HandleEM \\
Running & \overset{c}{=} & updatePosition.ID?pos \rightarrow getLOA.ID?loa \rightarrow computeSBI!loa?sbi \rightarrow \\
& & \quad \textbf{if } sbi \leq pos \\
& & \quad \textbf{then } applySB \rightarrow selectSpeed \rightarrow Running \\
& & \quad \textbf{else } releaseSB \rightarrow selectSpeed \rightarrow Running \\
HandleEM & \overset{c}{=} & receive.Warning.ID \rightarrow send.Ack.ID \rightarrow getPosition?pos \rightarrow getSBI?sbi \rightarrow \\
& & \quad \textbf{if } pos < sbi - OffsetDistance \\
& & \quad \textbf{then } indication.ID \rightarrow (driverAck.ID \rightarrow DriverResponsible \\
& & \qquad \Box\; EmergencyStop \\
& & \qquad \Box\; driverEB.ID \rightarrow EmergencyStop) \\
& & \quad \textbf{else } EmergencyStop \\
EmergencyStop & \overset{c}{=} & applyEB \rightarrow \mathtt{Skip} \\
DriverResponsible & \overset{c}{=} & \mathtt{Skip}
\end{array}$$

$$\begin{array}{l}
sbi : Position \\
standstillEB : Position \\
currentPosition : Position \\
currentSpeed : Speed \\
brakingMode : BrakingMode
\end{array}$$

Init

$$\begin{array}{l}
brakingMode = None \\
TargetSpeed < currentSpeed \leq MaxSpeed \\
currentPosition = StartPosition \\
position = StartPosition \\
sbi = StartSBI
\end{array}$$

com_applySB

$$\begin{array}{l}
\Delta(brakingMode) \\
\hline
\textbf{if } brakingMode = None \textbf{ then} \\
\quad brakingMode' = ServiceBrake \\
\textbf{else} \\
\quad brakingMode' = brakingMode
\end{array}$$

com_releaseSB

$$\begin{array}{l}
\Delta(brakingMode) \\
\hline
\textbf{if } brakingMode = ServiceBrake \textbf{ then} \\
\quad brakingMode' = None \\
\textbf{else} \\
\quad brakingMode' = brakingMode
\end{array}$$

com_selectSpeed

$$\begin{array}{l}
\Delta(currentSpeed) \\
\hline
\textbf{if } brakingMode = None \textbf{ then} \\
\quad TargetSpeed < currentSpeed' \leq MaxSpeed \\
\textbf{if } brakingMode = ServiceBrake \textbf{ then} \\
\quad currentSpeed' = TargetSpeed \\
\textbf{if } brakingMode = EmergencyBrake \textbf{ then} \\
\quad (currentSpeed' < TargetSpeed \;\wedge \\
\quad\; currentPosition + currentSpeed' \\
\qquad\quad \leq standstillEB) \vee \\
\quad (currentSpeed' = 0 \;\wedge \\
\quad\; currentPosition + TargetSpeed \\
\qquad\quad > standstillEB)
\end{array}$$

com_applyEB

$$\begin{array}{l}
\Delta(brakingMode, standstillEB) \\
\hline
brakingMode' = EmergencyBrake \\
\textbf{if } brakingMode = EmergencyBrake \textbf{ then} \\
\quad standstillEB' = standstillEB \\
\textbf{else} \\
\quad \textbf{if } currentSpeed = TargetSpeed \textbf{ then} \\
\qquad standstillEB' = currentPosition \\
\qquad\quad +StopDistance \\
\quad \textbf{else} \\
\qquad standstillEB' = currentPosition \\
\qquad\quad +TargetSpeedDistance + StopDistance
\end{array}$$

com_updatePosition

$\Delta(currentPosition)$
$pos! : Position$

$currentPosition' = currentPosition + currentSpeed$
$pos! = currentPosition'$

com_computeSBI

$\Delta(sbi)$
$loa?, sbi! : Position$

$sbi' = loa? - TargetSpeedDistance - StopDistance - MaxSpeed$
$sbi! = sbi'$

com_getPosition

$pos! : Position$

$pos! = currentPosition$

com_getSBI

$sbi! : Position$

$sbi! = sbi$

$\neg(\text{true} ^\frown \updownarrow updatePosition ^\frown updateBound > \ell ^\frown \updownarrow updatePosition)$
$\neg(\text{true} ^\frown (\boxminus updatePosition.ID \land updateBound < \ell))$
$\neg(\text{true} ^\frown \updownarrow receive.Warning.ID ^\frown (\boxminus applyEB \land \boxminus indication \land 0.5 < \ell) ^\frown (\boxminus indication \land 1 < \ell))$
$\neg(\text{true} ^\frown \updownarrow indication ^\frown (\boxminus driverAck \land \boxminus applyEB \land 5 < \ell))$

LeadingTrain$(ID : TrainID; StartPosition : Position)$

chan $send : [m! : Message, id : \{ID\}]$
chan $receive : [m? : Message, id : \{ID\}]$
chan $updatePosition : [id : \{ID\}, pos! : Position]$
chan $detectEmergency$
local_chan $selectSpeed, applyEB$

$\mathtt{main} \stackrel{c}{=} Running \,\square\, (detectEmergency \rightarrow send.Alert.ID \rightarrow applyEB \rightarrow selectSpeed \rightarrow \mathtt{main})$
$Running \stackrel{c}{=} updatePosition.ID \rightarrow selectSpeed \rightarrow \mathtt{main}$

$currentPosition : Position$
$currentSpeed : Speed$
$brakingMode : BrakingMode$

Init

$brakingMode = None$
$TargetSpeed \leq currentSpeed \leq MaxSpeed$
$position = StartPosition$

com_updatePosition

$\Delta(currentPosition)$
$pos! : Position$

$currentPosition' = currentPosition + currentSpeed$
$pos! = currentPosition'$

com_applyEB

$\Delta(brakingMode)$

$brakingMode' = EmergencyBrake$

com_selectSpeed

$\Delta(currentSpeed)$

if $brakingMode = None$ **then**
$TargetSpeed \leq currentSpeed'$
else
$currentSpeed' = 0$

$\neg(\text{true} ^\frown \updownarrow updatePosition ^\frown updateBound > \ell ^\frown \updownarrow updatePosition)$
$\neg(\text{true} ^\frown (\boxminus updatePosition.ID \land updateBound < \ell))$

RBC

chan $send : [m! : Message, id! : TrainID]$
chan $receive : [m? : Message, id? : TrainID]$

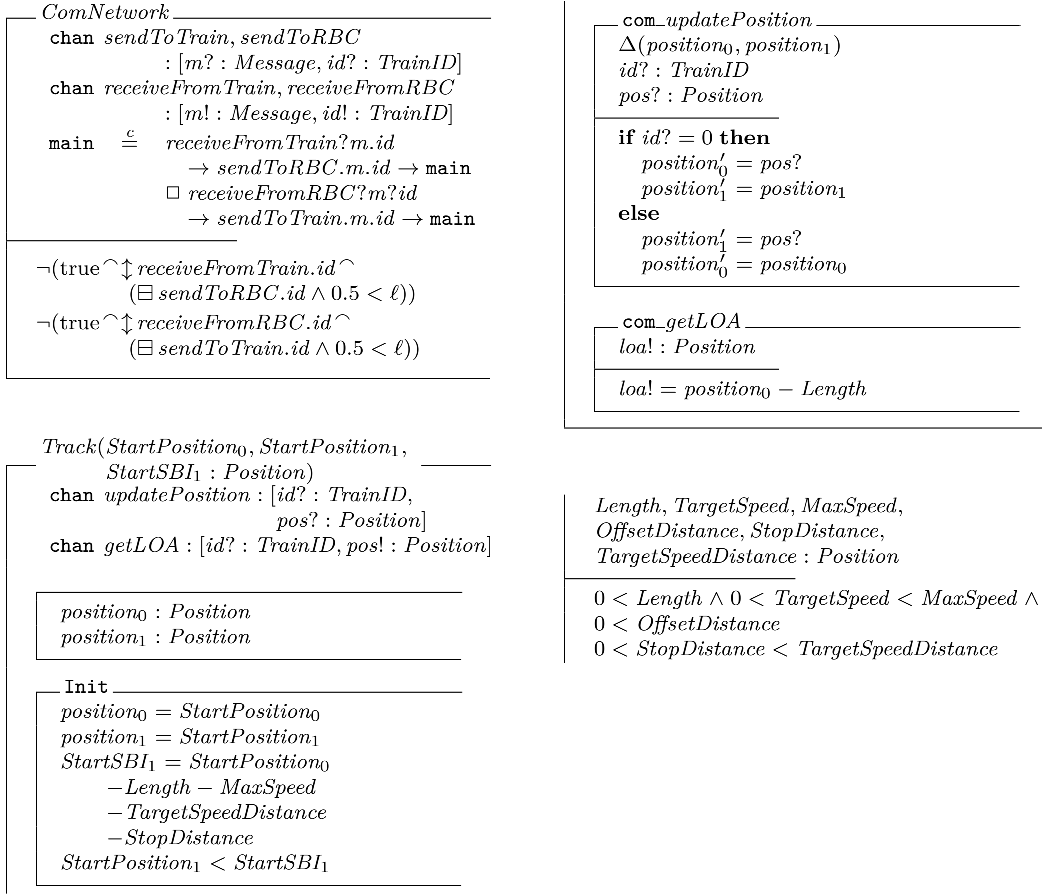
$\mathtt{main} \stackrel{c}{=} Idle$
$Idle \stackrel{c}{=} receive.Alert \rightarrow HandleEM$
$HandleEM \stackrel{c}{=} send.Warning!id \rightarrow receive.Ack.id \rightarrow Idle$

$\neg(\text{true} ^\frown \updownarrow receive.Alert ^\frown (\boxminus send.Warning \land 0.5 < \ell))$

Driver$(ID : TrainID)$

chan $indication : [id : \{ID\}]$
chan $driverAck, driverEB : [id : \{ID\}]$

$\mathtt{main} \stackrel{c}{=} indication.ID \rightarrow HandleEM$
$HandleEM \stackrel{c}{=} \mathtt{main} \sqcap driverAck.ID \rightarrow \mathtt{main} \sqcap driverEB.ID \rightarrow \mathtt{main}$



For abbreviated synchronisation alphabets and class instances

$$Trains == LT \mid\mid\mid RT$$

$$LT == LeadingTrain(0, StartPosition_0)$$

$$RT == RearTrain(1, StartPosition_1) \underset{D}{\parallel} Driver(1)$$

$$Track == Track(StartPosition_0, StartPosition_1, StartSBI_1)$$

$$A == updatePosition, getLOA$$

$$B == [send \mapsto receiveFromTrain, receive \mapsto sendToTrain]$$

$$C == [receiveFromRBC \mapsto send, sendToRBC \mapsto receive]$$

$$D == driverAck, driverEB, indication$$

the full case study model is defined by the CSP expression

$$\mathit{Trains} \underset{A}{\parallel} \mathit{Track} \underset{B}{\parallel} \mathit{ComNetwork} \underset{C}{\parallel} \mathit{RBC}.$$

Note that for synchronisation with the *ComNetwork* (communication alphabets B and C) the *linked parallel* operator [Ros98] is used. This is, instead of directly synchronising on the channels *send* and *receive* between the trains and the RBC, *send* and *receive* are mapped to *receiveFromTrain* and *sendToTrain*.

A.2.2. VA Proof Tree for the ETCS Case Study

This section contains the full proof tree for the ETCS VA from Sect. 8.2.4. In the proof tree, the rule (relation) abbreviates relational conversions in the theory of reals. We apply some of the rules implicitly: We waive the explicit application of the rules (weakening left) and (weakening right) and predominantly apply (process call) implicitly. We use the following abbreviations:

(next-step) = (prefix$_\delta$)(box step$_\delta$)(all right)(implication right)(and left)(or right)
(choice-split) = (box choice)(and right)

Main Part

$$\dfrac{\dfrac{\text{(assumption}_\Box\text{)}}{\mathit{Cons}, \psi \vdash [\mathit{ProcRec}]\Box \mathit{safe}} \text{(process step)}}{\mathit{Cons}, [\mathit{ProcFree}(\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running})]\psi \vdash [\mathit{ProcFree}(\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running})][\mathit{ProcRec}]\Box \mathit{safe}}$$

(3a)

$$\dfrac{\dfrac{(3a) \quad (3b)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{ProcFree}(\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running})][\mathit{ProcRec}]\Box \mathit{safe}} \text{(cut)}[\mathit{ProcFree}(\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running})]\psi}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{RTcurPos} + \mathit{SysLength} + \mathit{SysTrgSpdDst} + \mathit{SysStpDst} \vdash [\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running}][\mathit{ProcRec}]\Box \mathit{safe}} \text{(parallel uproc)}$$

(3)

$$\dfrac{\dfrac{(3b')}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{ProcFree}(\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running})]\Box \mathit{safe}}}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running}]\Box \mathit{safe}} \text{(parallel uproc)}$$

(2)

$$\cfrac{\cfrac{\cfrac{(2) \qquad (3)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [((\mathit{HandleEM} \underset{A_2}{\parallel} \mathit{Running}) \fatsemi \mathit{ProcRec})]\Box \mathit{safe}}\ (\text{sequence}_\Box)}{\mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\mathit{DetectEM}]\Box \mathit{safe}}\ (\text{process call})}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [(\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM}][\mathit{DetectEM}]\Box \mathit{safe}}\ (\text{box assumption}_\delta)$$

(1)

$$\cfrac{\cfrac{\cfrac{\text{assumption}_\Box}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [(\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM}]\Box \mathit{safe}} \qquad (1)}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [(\mathtt{Proc}_{\setminus A_0,\varnothing} \bullet \mathit{WaitEM} \fatsemi \mathit{DetectEM})]\Box \mathit{safe}}\ (\text{sequence}_\Box)}{\mathit{Cons}, \mathit{Init}, \mathit{LTcurPos} \geq \mathit{sf} \vdash [\text{ETCS-EM}]\Box \mathit{safe}}\ (\text{process call})$$

Proof Tree for Checking the δ-Branch

$$\cfrac{\begin{array}{c}(11)\\ \begin{array}{l} \mathit{Cons}, \mathit{LTcurPos} \geq \mathit{rd},\\ \mathit{RTcurSpd} \leq \mathit{SysTrgSpd},\\ \mathit{RTcurSpd} \leq \mathit{SysTrgSpd} \Rightarrow \mathit{RTstndEB} = \mathit{RTcurPos} + \mathit{SysStpDst},\\ \vdash\\ \mathit{RTstndEB} \leq \mathit{LTcurPos} - \mathit{SysLength} \wedge\\ \mathit{RTcurPos} \leq \mathit{RTstndEB} - \mathit{SysStpDst}\end{array}\end{array}}{\begin{array}{l} \mathit{Cons}, \mathit{LTcurPos} \geq \mathit{sf},\\ \mathit{RTcurSpd} \leq \mathit{SysTrgSpd},\\ \mathit{RTcurSpd} \leq \mathit{SysTrgSpd} \Rightarrow \mathit{RTstndEB} = \mathit{RTcurPos} + \mathit{SysStpDst},\\ \vdash\\ \mathit{RTstndEB} \leq \mathit{LTcurPos} - \mathit{SysLength} \wedge\\ \mathit{RTcurPos} \leq \mathit{RTstndEB} - \mathit{SysStpDst}\end{array}}\ (\text{relation})$$

(16)

$$\dfrac{\dfrac{\overset{\text{(axiom)}}{LTcurPos \geq sf \vdash RTcurPos + SysTrgSpdDst + SysStpDst \leq LTcurPos - SysLength} \quad \overset{\text{(relation)(axiom)}}{Cons \vdash RTcurPos \leq RTcurPos + SysTrgSpdDst}}{\begin{array}{l} Cons, LTcurPos \geq sf, \\ \vdash \\ RTcurPos + SysTrgSpdDst + SysStpDst \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTcurPos + SysTrgSpdDst \end{array}} \text{(and right)}}{\dfrac{\begin{array}{l} Cons, LTcurPos \geq sf, \\ RTstndEB = RTcurPos + SysTrgSpdDst + SysStpDst \\ \vdash \\ RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}}{\begin{array}{l} Cons, LTcurPos \geq sf, \\ RTcurSpd > SysTrgSpd, \\ RTcurSpd > SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysTrgSpdDst + SysStpDst \\ \vdash \\ RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}} \text{(implication left)(axiom)}} \text{(relation)}$$

(15)

$$\dfrac{(14) \quad \dfrac{(15) \qquad (16)}{\begin{array}{l} Cons, LTcurPos \geq sf, \\ RTcurSpd \leq SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysStpDst, \\ RTcurSpd > SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysTrgSpdDst + SysStpDst \\ \vdash \\ RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}} \text{(cut) } RTcurSpd > SysTrgSpd}{\dfrac{\begin{array}{l} Cons, LTcurPos \geq sf, RTbrkMd = BrkMdEmBrk, \\ RTcurSpd \leq SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysStpDst, \\ RTcurSpd > SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysTrgSpdDst + SysStpDst \\ \vdash \\ RTbrkMd = BrkMdEmBrk \wedge RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}}{Cons, LTcurPos \geq sf \vdash [RTappEB \bullet \varphi_1]\psi} \text{(next-step)}} \text{(and right)}$$

(13)

$$\dfrac{\dfrac{\dfrac{\overset{\text{(axiom)}}{LTcurPos \geq sf \vdash LTcurPos \geq sf, [P_8]\psi}}{LTcurPos \geq sf, LTcurPos < sf \vdash [P_8]\psi} \text{(relation)}}{\begin{array}{c} LTcurPos \geq sf, LTcurPos_{new} = LTcurPos, RTcurPos_{new} = RTcurPos, \\ LTcurPos_{new} < RTcurPos_{new} + SysLength + SysTrgSpdDst + SysStpDst \vdash [P_8]\psi \end{array}} \text{(relation)}}{LTcurPos \geq sf \vdash [P_2]\psi} \text{(next-event)}$$

(5)

$$\dfrac{\overset{\text{(axiom)}}{Cons, LTcurPos \geq rd, RTdrvRes_{new} \vdash RTdrvRes_{new}, RTbrkMd_{new} = \ldots}}{Cons, LTcurPos \geq rd \vdash [RTsetRes \bullet \varphi_2]\psi} \text{(next-step)}$$

(8)

$$\dfrac{\overset{\text{(axiom)}}{Cons, LTcurPos < sf \vdash LTcurPos < sf}}{Cons, LTcurPos < sf, LTcurPos \geq rd, LTcurPos \geq sf \vdash [RTappEB \bullet \varphi_1]\psi}\ \text{(relation)}$$

(10)

$$\begin{array}{l} Cons, LTcurPos \geq RTcurPos + SysLength \quad \text{(relation)(axiom)} \\ + SysStpDst, RTcurPos = RTstndEB - SysStpDst \vdash LTcurPos \geq RTstndEB + SysLength \end{array}$$

$$\dfrac{\overset{\text{(relation)(axiom)}}{RTcurPos = RTstndEB - SysStpDst \vdash RTcurPos \leq RTstndEB - SysStpDst}}{\begin{array}{l} Cons, LTcurPos \geq rd, \\ RTstndEB = RTcurPos + SysStpDst \\ \vdash \\ RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}}\ \text{(and right)}$$

(12)

$$\dfrac{\overset{\text{(axiom)}}{RTcurSpd \leq SysTrgSpd \vdash RTcurSpd \leq SysTrgSpd} \qquad (12)}{\begin{array}{l} Cons, LTcurPos \geq rd, \\ RTcurSpd \leq SysTrgSpd, \\ RTcurSpd \leq SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysStpDst, \\ \vdash \\ RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}}\ \text{(implication left)}$$

(11)

$$\overset{\text{(axiom)}}{RTbrkMd = BrkMdEmBrk \vdash RTbrkMd = BrkMdEmBrk}$$

(14)

$$\dfrac{\dfrac{(14) \qquad\qquad (11)}{\begin{array}{l} Cons, LTcurPos < sf, LTcurPos \geq rd, \\ RTcurSpd \leq SysTrgSpd, RTbrkMd = BrkMdEmBrk, \\ RTcurSpd \leq SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysStpDst, \\ RTcurSpd > SysTrgSpd \Rightarrow RTstndEB = RTcurPos + SysTrgSpdDst + SysStpDst \\ \vdash \\ RTbrkMd = BrkMdEmBrk \wedge RTstndEB \leq LTcurPos - SysLength \wedge \\ RTcurPos \leq RTstndEB - SysStpDst \end{array}}\ \text{(and right)}}{Cons, LTcurPos < sf, LTcurPos \geq rd, RTcurSpd \leq SysTrgSpd \vdash [RTappEB \bullet \varphi_1]\psi}\ \text{(next-step)}$$

(9)

$$\dfrac{(9) \qquad\qquad (10)}{\begin{array}{l} Cons, LTcurPos < sf, LTcurPos \geq rd, \\ (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [RTappEB \bullet \varphi_1]\psi \end{array}}\ \text{(or left)}$$

(7)

$$\dfrac{\dfrac{(13) \qquad (8)}{Cons, LTcurPos \geq sf \vdash [P_7]\psi}\ \text{(choice-split)(relation)}}{Cons, LTcurPos \geq sf \vdash [P_6]\psi}\ \text{(next-step)}$$

(6)

$$\cfrac{(7) \qquad (8)}{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_7]\psi}\ (\text{choice-split})$$

$$\cfrac{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_7]\psi}{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_6]\psi}\ (\text{next-step})$$

$$\cfrac{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_6]\psi}{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_4]\psi}\ (\text{process call})$$

$$\cfrac{Cons, LTcurPos < sf, LTcurPos \geq rd, (RTcurSpd \leq SysTrgSpd \vee LTcurPos \geq sf) \vdash [P_4]\psi}{\begin{array}{l} Cons, LTcurPos \geq sf, LTcurPos_{new} < sf_{new}, RTstndEB_{new} = RTstndEB, \\ RTdrvAck_{new} = RTdrvAck, LTcurPos_{new} \geq rd_{new}, \\ (RTcurSpd_{new} \leq SysTrgSpd \vee LTcurPos_{new} \geq sf_{new}) \vdash [P_{4,new}]\psi \end{array}}\ (\text{renaming})$$

$$\cfrac{\cdots \vdash [P_{4,new}]\psi}{Cons, LTcurPos \geq sf \vdash [P_3]\psi}\ (\text{next-step})$$

$$\cfrac{\overset{(6)}{Cons, LTcurPos \geq sf \vdash [P_6]\psi} \qquad \cfrac{Cons, LTcurPos \geq sf \vdash [P_3]\psi}{Cons, LTcurPos \geq sf \vdash [a \bullet \cdots \rightarrow P_3]\psi}\ (\text{next-step})}{\underset{(4)}{Cons, LTcurPos \geq sf \vdash [P_1]\psi}}\ (\text{box choice})(\text{and right})$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overset{(4)}{Cons, LTcurPos \geq sf \vdash [P_1]\psi} \qquad \overset{(5)}{Cons, LTcurPos \geq sf \vdash [P_2]\psi}}{Cons, LTcurPos \geq sf \vdash [P_1]\psi \wedge [P_2]\psi}\ (\text{and right})}{Cons, LTcurPos \geq sf \vdash [P_1 \ \Box\ P_2]\psi}\ (\text{box choice})}{Cons, LTcurPos \geq sf \vdash [P_0]\psi}\ (\text{process call})}{Cons, LTcurPos \geq sf \vdash \forall \Xi VR \Rightarrow [P_0]\psi}\ (\text{all right})(\text{implication right})(\text{renaming})}{Cons, LTcurPos \geq sf \vdash [LTdetEM \bullet \Xi VR][P_0]\psi}\ (\text{box step}_\delta)}{Cons, LTcurPos \geq sf \vdash [ProcFree(HandleEM \underset{A_2}{\parallel} Running)]\psi}\ (\text{prefix}_\delta)}{(3b)}$$

Proof Tree for Checking the □-Branch

In the proof of the □-branch of the tree, we use (∗) as abbreviation for derivations that already occur in the δ-branch. In addition, we use the following abbreviations:

$$\psi = \Box safe$$
$$\overline{\psi} = safe$$
$$\overline{VR} = (RTcurPos, LTcurPos)$$

$$\cfrac{(21) \qquad \cfrac{\overset{(\text{axiom})}{Cons, LTcurPos \geq sf \vdash LTcurPos \geq sf, [P_8]\psi}}{Cons, LTcurPos \geq sf \vdash [P_2][P_8]\psi}\ (*)}{\underset{(5')}{Cons, LTcurPos \geq sf \vdash [P_2]\psi}}\ (\text{prefix}_\Box)$$

$$\cfrac{(23) \qquad \cfrac{\overset{(\text{axiom})}{Cons, RTdrvRes \vdash RTdrvRes}}{[RTsetRes \bullet \varphi_2]\overline{\psi}}\ (\text{next-step})}{\underset{(8')}{Cons, LTcurPos \geq rd \vdash [RTsetRes \bullet \varphi_2]\psi}}\ (\text{box step})$$

$$\cfrac{(21) \qquad \cfrac{(21) \qquad (8')}{Cons, LTcurPos \geq sf \vdash [P_6]\psi}\ (\text{choice-split})(\text{relation})}{\underset{(6')}{Cons, LTcurPos \geq sf \vdash [P_6]\psi}}\ (\text{prefix}_\Box)$$

$$
\dfrac{\overset{(6')}{Cons, LTcurPos \geq sf \vdash [P_6]\psi} \qquad \dfrac{(21) \qquad \dfrac{\dfrac{(6') \qquad \dfrac{(22) \qquad \dfrac{\dfrac{(21) \qquad (8')}{Cons, LTcurPos \geq rd \vdash [P_6]\psi}\,(\text{choice-split})(\text{relation})}{Cons, LTcurPos \geq sf \vdash [b \bullet \ldots][P_4]\psi}\,(*)}{Cons, LTcurPos \geq sf \vdash [b \bullet \ldots]\psi}\,(\text{prefix}_\Box)}{Cons, LTcurPos \geq sf \vdash [P_3]\psi}\,(\text{choice-split})}{Cons, LTcurPos \geq sf \vdash [a \bullet \ldots][P_3]\psi}\,(\text{next-step})}{Cons, LTcurPos \geq sf \vdash [a \bullet \cdots \rightarrow P_3]\psi}\,(\text{prefix}_\Box)}{Cons, LTcurPos \geq sf \vdash [P_1]\psi}\,(\text{choice-split})
$$

(4′)

$$
\dfrac{\overset{(20)}{Cons, LTcurPos \geq sf \vdash safe} \qquad \dfrac{\overset{(20)}{Cons, LTcurPos \geq sf \vdash safe}}{Cons, LTcurPos \geq sf \vdash [LTdetEM \bullet \Xi VR]safe}\,(\text{next-step})}{Cons, LTcurPos \geq sf \vdash [LTdetEM \bullet \Xi VR]\Box safe}\,(\text{box step}_\delta)
$$

(19)

$$
\dfrac{\overset{(\text{axiom})}{Cons, LTcurPos \geq sf \vdash LTcurPos \geq sf}}{\begin{array}{l} Cons, RTcurPos \leq LTcurPos - SysTrgSpdDst - SysStpDst - SysLength \\ \vdash RTcurPos < LTcurPos - SysLength \vee RTdrvRes \end{array}}\,(\text{or right})(\text{relation})
$$

(20)

$$
\dfrac{(23) \qquad \dfrac{\overset{(\text{axiom})}{Cons, LTcurPos \geq rd \vdash LTcurPos \geq rd}}{Cons, LTcurPos \geq rd \vdash [\cdot \bullet \Xi\overline{VR}]\overline{\psi}}\,(\text{next-step})(\text{relation})}{Cons, LTcurPos \geq rd \vdash [\cdot \bullet \Xi\overline{VR}]\psi}\,(\text{box step})(\text{or right})(\text{relation})
$$

(21)

$$
\dfrac{\overset{(\text{axiom})}{Cons, LTcurPos \geq rd \vdash LTcurPos \geq rd}}{Cons, LTcurPos \geq rd \vdash \overline{\psi}}\,(\text{or right})(\text{relation})
$$

(23)

$$
\dfrac{\overset{(\text{relation})(23)}{Cons, LTcurPos \geq sf \vdash \overline{\psi}} \qquad \dfrac{\overset{(23)}{Cons, LTcurPos \geq rd \vdash \overline{\psi}}}{Cons, LTcurPos \geq sf \vdash [b \bullet \ldots]\overline{\psi}}\,(\text{next-step})}{Cons, LTcurPos \geq sf \vdash [b \bullet \ldots]\psi}\,(\text{box step})
$$

(22)

$$
\dfrac{\overset{(19)}{Cons, LTcurPos \geq sf \vdash [LTdetEM \bullet \Xi VR]\Box safe} \qquad \dfrac{(4') \qquad (5')}{Cons, LTcurPos \geq sf \vdash [LTdetEM \bullet \Xi VR][P_0]\Box safe}\,(*)}{Cons, LTcurPos \geq sf \vdash [ProcFree(HandleEM \underset{A_2}{\parallel} Running)]\Box safe}\,(\text{prefix}_\Box)
$$

(3b′)

Proof Tree for Checking the Timing Property

$$\text{(axiom)}$$

$$\frac{x_3 > y_3 \vdash x_3 > y_3}{y_3 \geq x_3, x_3 > y_3 \vdash \mathit{false}} \text{(relation)(negation left)}$$

$$\frac{y_3 \geq x_3, x_3 > y_3 \vdash \mathit{false}}{y_3 \geq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + y_3, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false}} \text{(relation)}$$

$$\frac{y_3 \geq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + y_3, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false}}{\begin{array}{c} y = y_3 + \mathit{len}_3, y_3 \geq 8, x \leq 8, x = x_3 + \mathit{len}_3, \mathit{len}_3 > 0, y_3 = \mathit{len}_2, \\ y_3 \leq 8, x_3 \leq 8, x_3 = \mathit{len}_0 + \mathit{len}_1 + \mathit{len}_2, \mathit{len}_2 > 0, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash \mathit{false} \end{array}} \text{(relation)(weakening left)}$$

$$\frac{\cdots}{y = \mathit{len}_2, y \leq 8, x \leq 8, x = \mathit{len}_0 + \mathit{len}_1 + \mathit{len}_2, \mathit{len}_2 > 0, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P_5^t]\mathit{false}} \text{(next-step)}$$

$$\frac{\cdots}{\begin{array}{c} y = \mathit{len}_2, y \leq 8, x \leq 8, x = x_2 + \mathit{len}_2, \mathit{len}_2 > 0, x_2 \leq 8, \\ x_2 = \mathit{len}_0 + \mathit{len}_1, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P_5^t]\mathit{false} \end{array}} \text{(relation)(weakening left)}$$

$$\frac{\cdots}{x \leq 8, x = \mathit{len}_0 + \mathit{len}_1, \mathit{len}_1 > 0, \mathit{len}_0 > 0 \vdash [P_3^t]\mathit{false}} \text{(next-step)}$$

$$\frac{\cdots}{x \leq 8, x = x_1 + \mathit{len}_1, \mathit{len}_1 > 0, x_1 < 8, x_1 = \mathit{len}_0, \mathit{len}_0 > 0 \vdash [P_3^t]\mathit{false}} \text{(relation)(weakening left)}$$

$$\frac{\cdots}{x < 8, x = \mathit{len}_1, \mathit{len}_1 > 0 \vdash [P_1^t]\mathit{false}} \text{(next-step)}$$

$$\frac{x < 8, x = \mathit{len}_1, \mathit{len}_1 > 0 \vdash [P_1^t]\mathit{false}}{\mathit{true} \vdash [P^t]\mathit{false}} \text{(next-step)}$$

A.2.3. Modified CSP-OZ-DC Specification Matching VA

$$Pos == \mathbb{R}$$
$$Spd == \mathbb{R}$$
$$Bool == \mathbb{R}$$
$$Msg == \mathbb{R}$$
$$BrkMd == \mathbb{R}$$
$$TrID == 0 \mathrel{..} 1$$

$SysLTID, SysRTID : TrID$

$SysLTID < SysRTID$

$MsgAck, MsgWrn, MsgAlrt : \mathbb{R}$

$MsgAck < MsgWrn$
$MsgWrn < MsgAlrt$

$BrkMdNone, BrkMdSrvBrk,$
$BrkMdEmBrk : \mathbb{R}$

$BrkMdNone < BrkMdSrvBrk$
$BrkMdSrvBrk < BrkMdEmBrk$

$SysMaxSpd, SysTrgSpd : Spd$

$SysTrgSpd > 0$
$SysTrgSpd < SysMaxSpd$

$SysLength, SysStpDst, SysTrgSpdDst,$
$SysLTStrtPos, SysRTStrtPos,$
$SysRTStrtSBI,$
$SysLTStrtPos, SysOffDst : Pos$

$SysLength > 0$
$SysStpDst > 0$
$SysStpDst < SysTrgSpdDst$
$SysRTStrtSBI = SysLTStrtPos$
$\quad -SysLength - SysMaxSpd$
$\quad -SysTrgSpdDst - SysStpDst$
$SysRTStrtPos < SysRTStrtSBI$
$SysOffDst > 0$
$SysMaxSpd = 1$
$SysTrgSpdDst = 10$

RT

```
method posGEsbi
method posLTsbi
method RTappEB
method RTappSB
method RTcmpSBI
method RTgetLOA : [loa? : Pos]
method RTgetPos
method RTgetSBI
method RTrelSB
method RTselSpd
method RTsetRes
method RTupPos : [rpos! : Pos]
chan CNrecFT : [i! : TrID; m! : Msg]
chan CNsndTT : [i? : TrID; m? : Msg]
chan RTdrvAck
chan RTdrvEB
chan RTind
```

$RTcurSpd : Spd$
$RTbrkMd : BrkMd$
$RTCSPHsbi : Pos$
$RTCSPHpos : Pos$
$RTdrvRes : Bool$
$RTsbi : Pos$
$RTcurPos : Pos$
$RTCSPRloa : Pos$
$RTstndEB : Pos$
$RTCSPRsbi : Pos$

Init

$RTcurPos = SysRTStrtPos$
$SysTrgSpd < RTcurSpd$
$RTcurSpd \leq SysMaxSpd$
$RTsbi = SysRTStrtSBI$
$RTbrkMd = BrkMdNone$

enable_CNsndTT

$i? : TrID;\ m? : Msg$

effect_CNsndTT

$\Delta(RTCSPm)$
$i? : TrID;\ m? : Msg$

$i? = SysRTID$
$m? = MsgWrn$

effect_$RTappEB$

$\Delta(RTbrkMd, RTstndEB)$

$RTbrkMd' = BrkMdEmBrk$
$RTcurSpd \leq SysTrgSpd \Rightarrow$
$\quad RTstndEB' = RTcurPos + SysStpDst$
$RTcurSpd > SysTrgSpd \Rightarrow$
$\quad RTstndEB' = RTcurPos$
$\quad\quad + SysTrgSpdDst + SysStpDst$

effect_$RTgetPos$

$\Delta(RTCSPHpos)$

$RTCSPHpos' = RTcurPos$

enable_$CSPposGEsbi$

$RTCSPHpos \geq RTCSPHsbi$
$\quad - SysOffDst$

effect_$RTcmpSBI$

$\Delta(RTsbi, RTCSPRsbi)$

$RTsbi' = RTCSPRloa$
$\quad - SysTrgSpdDst - SysStpDst$
$\quad - SysMaxSpd$
$RTCSPRsbi' = RTsbi'$

enable_$RTrelSB$

$RTCSPRsbi > RTcurPos$

effect_$RTrelSB$

$\Delta(RTbrkMd)$

$RTbrkMd < BrkMdEmBrk$
$\quad \Rightarrow RTbrkMd' = BrkMdNone$
$RTbrkMd \geq BrkMdEmBrk$
$\quad \Rightarrow RTbrkMd' = RTbrkMd$

effect_$RTgetSBI$

$\Delta(RTCSPHsbi)$

$RTCSPHsbi' = RTsbi$

enable_$RTupPos$

$rpos! : Pos$

$RTcurSpd \leq SysMaxSpd$

effect_$RTupPos$

$\Delta(RTcurPos)$
$rpos! : Pos$

$RTcurPos' = RTcurPos + RTcurSpd$
$RTcurPos' = rpos!$

enable_$RTappSB$

$RTCSPRsbi \leq RTcurPos$

effect_$RTappSB$

$\Delta(RTbrkMd)$

$RTbrkMd > BrkMdNone$
$\quad \Rightarrow RTbrkMd' = RTbrkMd$
$RTbrkMd \leq BrkMdNone$
$\quad \Rightarrow RTbrkMd' = BrkMdSrvBrk$

effect_$RTsetRes$

$\Delta(RTdrvRes)$

$RTdrvRes' = 1$

enable_$RTgetLOA$

$loa? : Pos$

effect_$RTgetLOA$

$\Delta(RTCSPRloa)$
$loa? : Pos$

$RTCSPRloa' = loa?$

enable_$CNrecFT$

$i! : TrID; \; m! : Msg$

effect_$CNrecFT$

$i! : TrID; \; m! : Msg$

$(i! = SysLTID \wedge m! = MsgAlrt) \vee$
$(i! = SysRTID \wedge m! = MsgAck)$

effect_$RTselSpd$

$\Delta(RTcurSpd)$

$RTbrkMd \leq BrkMdNone$
$\quad \Rightarrow (RTcurSpd' > SysTrgSpd \wedge$
$\quad\quad RTcurSpd' \leq SysMaxSpd)$
$(RTbrkMd > BrkMdNone \wedge$
$\quad\quad RTbrkMd < BrkMdEmBrk)$
$\quad \Rightarrow RTcurSpd' = SysTrgSpd$
$RTbrkMd \geq BrkMdEmBrk \Rightarrow$
$\quad ((RTcurSpd' < SysTrgSpd \wedge$
$\quad\quad RTcurPos + SysTrgSpd$
$\quad\quad\quad \leq RTstndEB) \vee$
$\quad (RTcurSpd' = 0 \wedge$
$\quad\quad RTcurPos + SysTrgSpd$
$\quad\quad\quad > RTstndEB))$

enable_$posLTsbi$

$RTCSPHpos < RTCSPHsbi - SysOffDst$

$\neg\Diamond(\updownarrow RTind ^\frown (\boxminus RTsetRes \land \boxminus RTappEB \land \ell > 1))$
$\neg\Diamond(\updownarrow RTupPos ^\frown (\ell < 1) ^\frown \updownarrow RTupPos)$
$\neg\Diamond(\updownarrow CNsndTT ^\frown (\boxminus RTappEB \land \boxminus RTind \land (\ell > 1)) ^\frown \updownarrow RTind ^\frown (\boxminus RTind \land (\ell > 1)))$

LT

method *LTappEB*
method *LTdetEM*
method *LTgetPos* : [*gpos*! : *Pos*]
method *LTselSpd*
method *LTupPos* : [*lpos*! : *Pos*]
chan *CNrecFT* : [*i*! : *TrID*; *m*! : *Msg*]
chan *CNsndTT* : [*i*? : *TrID*; *m*? : *Msg*]

LTbrkMd : *BrkMd*
LTcurSpd : *Spd*
LTcurPos : *Pos*

Init
$LTcurPos = SysLTStrtPos$
$SysTrgSpd \leq LTcurSpd$
$LTcurSpd \leq SysMaxSpd$
$LTbrkMd = BrkMdNone$

enable_*CNsndTT*
i? : *TrID*; *m*? : *Msg*

effect_*CNsndTT*
i? : *TrID*; *m*? : *Msg*

effect_*LTappEB*
$\Delta(LTbrkMd)$

$LTbrkMd' = BrkMdEmBrk$

enable_*LTgetPos*
gpos! : *Pos*

effect_*LTgetPos*
gpos! : *Pos*

$gpos! = LTcurPos$

enable_*LTupPos*
lpos! : *Pos*

effect_*LTupPos*
$\Delta(LTcurPos)$
lpos! : *Pos*

$LTcurPos' = LTcurPos + LTcurSpd$
$lpos! = LTcurPos'$

enable_*CNrecFT*
i! : *TrID*; *m*! : *Msg*

effect_*CNrecFT*
i! : *TrID*; *m*! : *Msg*

$(i! = SysLTID \land m! = MsgAlrt) \lor (i! = SysRTID \land m! = MsgAck)$

effect_*LTselSpd*
$\Delta(LTcurSpd)$

$LTbrkMd \leq BrkMdNone \Rightarrow (SysTrgSpd \leq LTcurSpd' \land LTcurSpd' \leq SysMaxSpd)$
$LTbrkMd > BrkMdNone \Rightarrow LTcurSpd' = 0$

$\neg\Diamond((\boxminus LTupPos \land (\ell > 1)))$
$\neg\Diamond(\updownarrow LTdetEM ^\frown (\boxminus CNrecFT \land (\ell > 1)))$

RTDrv

method *RTdrvAck*
method *RTdrvEB*
method *RTind*

$$RTDrvHndlEM \stackrel{c}{=} RTdrvAck \rightarrow \mathtt{main} \sqcap RTdrvEB \rightarrow \mathtt{main} \sqcap \mathtt{main}$$
$$\mathtt{main} \stackrel{c}{=} RTind \rightarrow RTDrvHndlEM$$

Trck

chan *CNrecFT* : [*i*! : *TrID*; *m*! : *Msg*]
chan *CNsndTT* : [*i*? : *TrID*; *m*? : *Msg*]
chan *posGEsbi*
chan *posLTsbi*
chan *LTappEB*
chan *LTdetEM*
chan *LTgetPos* : [*gpos*? : *Pos*]
chan *LTselSpd*
chan *LTupPos* : [*lpos*? : *Pos*]
chan *RTappEB*
chan *RTappSB*
chan *RTcmpSBI*
chan *RTdrvAck*
chan *RTdrvEB*
chan *RTgetLOA* : [*loa*! : *Pos*]
chan *RTgetPos*
chan *RTgetSBI*
chan *RTind*
chan *RTrelSB*
chan *RTselSpd*
chan *RTsetRes*
chan *RTupPos* : [*rpos*? : *Pos*]

$$
\begin{array}{rcl}
\mathtt{main} & \stackrel{c}{=} & \mathit{Running}_0 \mathbin{;} \mathit{Emergency}_C \\
 & ; & \mathit{Running}_2 \\
\mathit{Emergency}_C & \stackrel{c}{=} & (\mathit{HandleEM}_C \mathbin{\underset{A}{\|}} \mathit{Running}_C) \\
\mathit{Running}_C & \stackrel{c}{=} & ((\mathit{LTdetEM} \rightarrow \mathit{Running}_1) \\
 & ; & ((\mathit{RTappEB} \rightarrow \mathtt{Skip}) \\
 & \Box & (\mathit{RTsetRes} \rightarrow \mathtt{Skip}))) \\
A & = & \{\mathit{LTdetEM}, \mathit{RTappEB}, \\
 & & \mathit{RTsetRes}\}
\end{array}
$$

[ProcWaitEM]

$$
\begin{array}{rcl}
\mathit{Running}_0 & \stackrel{c}{=} & \mathit{RTRunning}_0 \mathbin{|||} \mathit{LTRunning}_0 \\
\mathit{RTRunning}_0 & \stackrel{c}{=} & ((\mathit{RTupPos} \rightarrow \mathit{RTgetL0}) \\
 & \Box & (\mathit{RTupPos} \rightarrow \mathtt{Skip})) \\
\mathit{RTgetL0} & \stackrel{c}{=} & (\mathit{RTgetLOA} \rightarrow \mathit{State3}) \\
\mathit{State3} & \stackrel{c}{=} & (\mathit{RTcmpSBI} \rightarrow \mathit{State4}) \\
\mathit{State4} & \stackrel{c}{=} & ((\mathit{RTappSB} \rightarrow \mathit{State5}) \\
 & \Box & (\mathit{RTrelSB} \rightarrow \mathit{State5})) \\
\mathit{State5} & \stackrel{c}{=} & (\mathit{RTselSpd} \rightarrow \mathit{RTRunning}_0) \\
\mathit{LTRunning}_0 & \stackrel{c}{=} & ((\mathit{LTupPos} \rightarrow \mathit{LTextch0}) \\
 & \Box & (\mathit{LTupPos} \rightarrow \mathtt{Skip})) \\
\mathit{LTextch0} & \stackrel{c}{=} & (\mathit{LTselSpd} \rightarrow \mathit{LTRunning}_0)
\end{array}
$$

[ProcAckTime]

$$
\begin{array}{rcl}
\mathit{HandleEM}_C & \stackrel{c}{=} & (\mathit{Detect} \mathbin{;} \mathit{RTsend}) \\
\mathit{Detect} & \stackrel{c}{=} & (\mathit{LTdetEM} \rightarrow \\
 & & \quad \mathit{LTsndAlrt}) \\
\mathit{LTsndAlrt} & \stackrel{c}{=} & (\mathit{CNrecFT} \rightarrow \mathtt{Skip}) \\
\mathit{RTsend} & \stackrel{c}{=} & (\mathit{CNsndTT} \rightarrow \mathit{RTrec}) \\
\mathit{RTrec} & \stackrel{c}{=} & (\mathit{CNrecFT} \rightarrow \mathit{RTgetP}) \\
\mathit{RTgetP} & \stackrel{c}{=} & (\mathit{RTgetPos} \rightarrow \mathit{RTgetS}) \\
\mathit{RTgetS} & \stackrel{c}{=} & (\mathit{RTgetSBI} \rightarrow \\
 & & \quad \mathit{RTposOPsbi}) \\
\mathit{RTposOPsbi} & \stackrel{c}{=} & ((\mathit{posGEsbi} \rightarrow \mathit{RTStop}) \\
 & \Box & (\mathit{posLTsbi} \rightarrow \mathit{RTind})) \\
\mathit{RTind} & \stackrel{c}{=} & (\mathit{RTind} \rightarrow \mathit{RTchoice}) \\
\mathit{RTchoice} & \stackrel{c}{=} & ((\mathit{RTdrvAck} \rightarrow \mathit{State0}) \\
 & \sqcap & (\mathit{RTdrvEB} \rightarrow \mathit{RTStop}) \\
 & \sqcap & \mathit{RTStop}) \\
\mathit{RTStop} & \stackrel{c}{=} & (\mathit{RTappEB} \rightarrow \mathtt{Skip}) \\
\mathit{State0} & \stackrel{c}{=} & (\mathit{RTsetRes} \rightarrow \mathtt{Skip})
\end{array}
$$

[ProcRec]

$$
\begin{array}{rcl}
\mathit{Running}_2 & \stackrel{c}{=} & \mathit{RTRunning}_2 \mathbin{|||} \mathit{LTRunning}_2 \\
\mathit{RTRunning}_2 & \stackrel{c}{=} & (\mathit{RTgetLOA} \rightarrow \mathit{RTcomp2}) \\
\mathit{RTcomp2} & \stackrel{c}{=} & (\mathit{RTcmpSBI} \rightarrow \mathit{RTapprel2}) \\
\mathit{RTapprel2} & \stackrel{c}{=} & ((\mathit{RTappSB} \rightarrow \mathit{RTselect2}) \\
 & \Box & (\mathit{RTrelSB} \rightarrow \mathit{RTselect2})) \\
\mathit{RTselect2} & \stackrel{c}{=} & (\mathit{RTselSpd} \rightarrow \mathit{RTRun2}) \\
\mathit{RTRun2} & \stackrel{c}{=} & (\mathit{RTupPos} \rightarrow \mathit{RTRunning}_2) \\
\mathit{LTRunning}_2 & \stackrel{c}{=} & (\mathit{LTselSpd} \rightarrow \mathit{LTmain2}) \\
\mathit{LTmain2} & \stackrel{c}{=} & (\mathit{LTupPos} \rightarrow \mathit{LTRunning}_2)
\end{array}
$$

[ProcRun]

$$
\begin{array}{rcl}
\mathit{Running}_1 & \stackrel{c}{=} & \mathit{RTRunning}_1 \mathbin{|||} \mathit{LTRunning}_1 \\
\mathit{RTRunning}_1 & \stackrel{c}{=} & (\mathit{LTgetPos} \rightarrow \mathit{RTgetL1}) \\
\mathit{RTgetL1} & \stackrel{c}{=} & (\mathit{RTgetLOA} \rightarrow \mathit{RTcomp1}) \\
\mathit{RTcomp1} & \stackrel{c}{=} & (\mathit{RTcmpSBI} \rightarrow \mathit{RTapprel1}) \\
\mathit{RTapprel1} & \stackrel{c}{=} & ((\mathit{RTappSB} \rightarrow \mathit{RTselect1}) \\
 & \Box & (\mathit{RTrelSB} \rightarrow \mathit{RTselect1})) \\
\mathit{RTselect1} & \stackrel{c}{=} & (\mathit{RTselSpd} \rightarrow \mathit{RTRun1}) \\
\mathit{RTRun1} & \stackrel{c}{=} & ((\mathit{RTupPos} \rightarrow \mathit{RTgetL1}) \\
 & \Box & (\mathit{RTupPos} \rightarrow \mathtt{Skip})) \\
\mathit{LTRunning}_1 & \stackrel{c}{=} & (\mathit{LTappEB} \rightarrow \mathit{LTselect1}) \\
\mathit{LTselect1} & \stackrel{c}{=} & (\mathit{LTselSpd} \rightarrow \mathit{LTmain1}) \\
\mathit{LTmain1} & \stackrel{c}{=} & ((\mathit{LTupPos} \rightarrow \mathit{LTextch1}) \\
 & \Box & (\mathit{LTupPos} \rightarrow \mathtt{Skip})) \\
\mathit{LTextch1} & \stackrel{c}{=} & (\mathit{LTselSpd} \rightarrow \mathit{LTmain1})
\end{array}
$$

$\mathit{TrRTpos} : \mathit{Pos}$
$\mathit{TrLTpos} : \mathit{Pos}$

Init

$\mathit{TrLTpos} = \mathit{SysLTStrtPos}$
$\mathit{TrRTpos} = \mathit{SysRTStrtPos}$

enable_$\mathit{CNsndTT}$

$i? : \mathit{TrID};\ m? : \mathit{Msg}$

effect_$\mathit{CNsndTT}$

$i? : \mathit{TrID};\ m? : \mathit{Msg}$

enable_$\mathit{LTgetPos}$

$\mathit{gpos}? : \mathit{Pos}$

effect_$\mathit{LTgetPos}$

$\Delta(\mathit{TrLTpos})$
$\mathit{gpos}? : \mathit{Pos}$

$\mathit{TrLTpos}' = \mathit{gpos}?$

enable_$\mathit{RTupPos}$

$\mathit{rpos}? : \mathit{Pos}$

effect_$\mathit{RTupPos}$

$\Delta(\mathit{TrRTpos})$
$\mathit{rpos}? : \mathit{Pos}$

$\mathit{TrRTpos}' = \mathit{rpos}?$

enable_$\mathit{RTgetLOA}$

$\mathit{loa}! : \mathit{Pos}$

effect_$\mathit{RTgetLOA}$

$\mathit{loa}! : \mathit{Pos}$

$\mathit{loa}! = \mathit{TrLTpos} - \mathit{SysLength}$

enable_LTupPos

$lpos? : Pos$

effect_LTupPos

$\Delta(TrLTpos)$
$lpos? : Pos$

$TrLTpos' = lpos?$

enable_CNrecFT

$i! : TrID;\ m! : Msg$

effect_CNrecFT

$i! : TrID;\ m! : Msg$

RBC

chan $CNrecFRwrn : [i! : TrID;\ m! : Msg]$
chan $CNsndTRack : [i? : TrID;\ m? : Msg]$
chan $CNsndTRalrt : [i? : TrID;\ m? : Msg]$

$RBCHndlEM \overset{c}{=} CNrecFRwrn \rightarrow RBCsnd$
$RBCsnd \overset{c}{=} CNsndTRack \rightarrow \mathtt{main}$
$\mathtt{main} \overset{c}{=} CNsndTRalrt \rightarrow RBCHndlEM$

enable_CNsndTRack

$i? : TrID;\ m? : Msg$

effect_CNsndTRack

$i? : TrID;\ m? : Msg$

$i? = SysRTID$
$m? = MsgAck$

enable_CNsndTRalrt

$i? : TrID;\ m? : Msg$

effect_CNsndTRalrt

$\Delta(RBCCSPi)$
$i? : TrID;\ m? : Msg$

$i? = SysLTID$
$m? = MsgAlrt$

enable_CNrecFRwrn

$i! : TrID;\ m! : Msg$

effect_CNrecFRwrn

$\Delta(RBCCSPi, RBCCSPm)$
$i! : TrID;\ m! : Msg$

$i! = SysRTID$
$m! = MsgWrn$

$\neg\Diamond(\updownarrow CNsndTRalrt ^\frown \boxminus CNrecFRwrn \wedge \ell > 1)$

CN

method $CNrecFRwrn : [i? : TrID;\ m? : Msg]$
method $CNrecFT : [i? : TrID;\ m? : Msg]$
method $CNsndTRack : [i! : TrID;\ m! : Msg]$
method $CNsndTRalrt : [i! : TrID;\ m! : Msg]$
method $CNsndTT : [i! : TrID;\ m! : Msg]$

$CNrecFromRBC \overset{c}{=} CNsndTT \rightarrow \mathtt{main}$
$CNrecFromTr \overset{c}{=} CNsndTRack \rightarrow \mathtt{main}\ \square\ CNsndTRalrt \rightarrow \mathtt{main}$
$\mathtt{main} \overset{c}{=} CNrecFRwrn \rightarrow CNrecFromRBC\ \square\ CNrecFT \rightarrow CNrecFromTr$

$CNi : TrID$
$CNm : Msg$

enable_CNsndTT

$i! : TrID;\ m! : Msg$

effect_CNsndTT

$i! : TrID;\ m! : Msg$

$m! = CNm$
$i! = CNi$

enable_CNsndTRack

$i! : TrID;\ m! : Msg$

effect_CNsndTRack

$i! : TrID;\ m! : Msg$

$m! = CNm$
$i! = CNi$

enable_CNsndTRalrt

$i! : TrID;\ m! : Msg$

effect_CNsndTRalrt

$i! : TrID;\ m! : Msg$

$m! = CNm$
$i! = CNi$

enable_CNrecFT

$i? : TrID;\ m? : Msg$

effect_CNrecFT

$\Delta(CNi, CNm)$
$i? : TrID;\ m? : Msg$

$CNi' = i?$
$CNm' = m?$

enable_CNrecFRwrn

$i? : TrID;\ m? : Msg$

effect_*CNrecFRwrn*

$\Delta(CNi, CNm)$
$i? : TrID;\ m? : Msg$

$CNi' = i?$
$CNm' = m?$

$\neg\Diamond(\updownarrow CNrecFT \frown (\boxminus CNsndTRalrt$
$\quad \wedge \boxminus CNsndTRack \wedge \ell > 1))$
$\neg\Diamond(\updownarrow CNrecFRwrn \frown (\boxminus CNsndTT \wedge \ell > 1))$

Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AdBO09] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs, 3rd Edition.* Texts in Computer Science. Springer, Heidelberg, 2009.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AM98] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer, Heidelberg, 1998.

[Art96] R. Arthan. Arithmetic for Z. Technical report, FMU/IED/WRK055, ICL, Eskdale Road, Winnersh, Berks, 1996.

[Bal05] M. Balser. *Verifying Concurrent Systems with Symbolic Execution.* PhD thesis, Fakultät für Angewandte Informatik, Universität Augsburg, 2005.

[BDFW07] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In F. Arbab and M. Sirjani, editors, *FSEN 2007*, volume 4767 of *Lecture Notes in Computer Science*, pages 17–32. Springer, Heidelberg, 2007.

[BDFW08] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundamenta Informaticae*, 89(4):369–392, 2008.

[BG03] S. Burmester and H. Giese. The Fujaba real-time statechart plugin. In *Proceedings of the first International Fujaba Days 2003, Kassel, Germany*, 2003.

[BGH+05] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE 2005*, pages 670–671. ACM, New York, NY, 2005.

[BH07] R. Bubel and R. Hähnle. Pattern-driven formal specification. In *Verification of Object-Oriented Software. The KeY Approach*, chapter 6. Springer, Heidelberg, 2007.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2007.

[BLR95] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In P. Wolper, editor, *CAV 1995*, volume 939 of *Lecture Notes in Computer Science*, pages 196–210. Springer, Heidelberg, 1995.

[BMW06] I. Brückner, B. Metzler, and H. Wehrheim. Optimizing slicing of formal specifications by deductive verification. *Nordic Journal of Computing*, 13(1-2):22–45, 2006.

[Brü04] I. Brückner. Slicing CSP-OZ Specifications. In P. Pettersson and W. Yi, editors, *Proceedings of the 16th Nordic Workshop on Programming Theory*, number 2004-041 in Technical Reports of the Department of Information Technology, pages 71–73. Uppsala University, 2004.

[Brü07] I. Brückner. Slicing Concurrent Real-Time System Specifications for Verification. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *Lecture Notes in Computer Science*, pages 54–74. Springer, Heidelberg, 2007.

[Brü08a] I. Brückner. Slicing integrated formal specifications for verification. Reports of the Department for Computing Science 2/08, University of Oldenburg, 2008.

[Brü08b] I. Brückner. *Slicing Integrated Formal Specifications for Verification.* PhD thesis, University of Paderborn, 2008.

[But92] M. J. Butler. *A CSP Approach To Action Systems.* PhD thesis, University of Oxford, 1992.

[But00] M. J. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.

[BW05] I. Brückner and H. Wehrheim. Slicing an integrated formal method for verification. In K.-K. Lau and R. Banach, editors, *ICFEM 2005*, volume

3785 of *Lecture Notes in Computer Science*, pages 360–374. Springer, Heidelberg, 2005.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 353–361. IEEE Computer Society Press, 1989.

[DHO06] W. Damm, H. Hungar, and E.-R. Olderog. Verification of cooperating traffic agents. *International Journal of Control*, 79(5):395 – 421, 2006.

[DHQ+04] J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi. Timed patterns: TCOZ to timed automata. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 483–498. Springer, Heidelberg, 2004.

[DKFW10] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 271–274. Springer, Heidelberg, 2010.

[DL02] H. Dierks and M. Lettrari. Constructing test automata from graphical real-time requirements. In W. Damm and E.-R. Olderog, editors, *FTRTFT 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 433–453. Springer, Heidelberg, 2002.

[DL09] L. D'Errico and M. Loreti. Assume-Guarantee Verification of Concurrent Systems. In J. Field and V. T. Vasconcelos, editors, *COORDINATION 2009*, volume 5521 of *Lecture Notes in Computer Science*, pages 288–305. Springer, Heidelberg, 2009.

[DL10] L. D'Errico and M. Loreti. Property-preserving refinement of concurrent systems. In M. Wirsing, M. Hofmann, and A. Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 222–236. Springer, Heidelberg, 2010.

[DMO+07] W. Damm, A. Mikschl, J. Oehlerking, E.-R. Olderog, J. Pang, A. Platzer, M. Segelken, and B. Wirtz. Automating verification of cooperation, control, and design in traffic applications. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 115–169. Springer, Heidelberg, 2007.

[Dou97] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.

[Dou02] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

[Dou04] B. P. Douglass. *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 2004.

[dRdBH+01] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[dRE98] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[DW03] J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 127–147. Springer, Heidelberg, 2003.

[ECS99] ECSAG. ERTMS/ETCS Functional requirements specification. `http://www.era.europa.eu/`, 1999. Version 4.29.

[EF82] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982.

[ERT02] ERTMS User Group, UNISIG. ERTMS/ETCS System requirements specification. `http://www.era.europa.eu/`, 2002. Version 2.2.2.

[Fab05] J. Faber. Verifying real-time aspects of the European Train Control System. In *Proceedings of the 17th Nordic Workshop on Programming Theory*, pages 67–70. University of Copenhagen, 2005.

[Fab09] J. Faber. Verification architectures for real-time systems. In M. Mousavi and E. Sekerinski, editors, *Proceedings of Formal Methods 2009 Doctoral Symposium*, number 09-15 in CS-Report, pages 14–19. Eindhoven University of Technology, 2009.

[Fab10a] J. Faber. Verification Architectures: Compositional reasoning for real-time systems. Reports of SFB/TR 14 AVACS 65, SFB/TR 14 AVACS, 2010. `http://www.avacs.org`.

[Fab10b] J. Faber. Verification Architectures: Compositional reasoning for real-time systems. In D. Méry and S. Merz, editors, *IFM 2010*, volume 6396 of *Lecture Notes in Computer Science*, pages 136–151. Springer, Heidelberg, 2010.

[FCA07] A. Flores, A. Cechich, and G. Aranda. A generic model of object-oriented patterns specified in RSL. In T. Taibi, editor, *Design Patterns Formalization Techniques*, chapter 3, pages 44–72. IGI Publishing, 2007.

[FH07] M. Fränzle and M. R. Hansen. Deciding an interval logic with accumulated durations. In *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 201–215. Springer, Heidelberg, 2007.

[FIJSS10a] J. Faber, C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. Automatic verification of parametric specifications with complex topologies. In D. Méry and S. Merz, editors, *IFM 2010*, volume 6396 of *Lecture Notes in Computer Science*, pages 152–167. Springer, Heidelberg, 2010.

[FIJSS10b] J. Faber, C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. Automatic verification of parametric specifications with complex topologies. Reports of SFB/TR 14 AVACS 66, SFB/TR 14 AVACS, 2010. `http://www.avacs.org`.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *FMOODS 1997*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, University of Oldenburg, 2000.

[FJSS07] J. Faber, S. Jacobs, and V. Sofronie-Stokkermans. Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *Lecture Notes in Computer Science*, pages 233–252. Springer, Heidelberg, 2007.

[FLOQ11] J. Faber, S. Linker, E.-R. Olderog, and J.-D. Quesel. Syspect—modelling, specifying, and verifying real-time systems with rich data. *International Journal of Software and Informatics*, 5(1-2):117 – 137, 2011.

[FM06] J. Faber and R. Meyer. Model checking data-dependent real-time properties of the European Train Control System. In *Formal Methods in Computer Aided Design*, pages 76–77. IEEE Computer Society, 2006.

[FMS98] B. Finkbeiner, Z. Manna, and H. Sipma. Deductive verification of modular systems. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 239–275. Springer, Heidelberg, 1998.

[FPS08] B. Finkbeiner, H.-J. Peter, and S. Schewe. Synthesizing certificates in networks of timed automata. In *RTSS 2008*, pages 183–194. IEEE Computer Society, Los Alamitos, CA, 2008.

[FPS10] B. Finkbeiner, H.-J. Peter, and S. Schewe. Synthesising certificates in networks of timed automata. *IET Software*, 4(3):222–235, 2010.

[Frä04] M. Fränzle. Model-checking dense-time Duration Calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[FS07] J. Faber and I. Stierand. From high-level verification to real-time scheduling: A property-preserving integration. Reports of SFB/TR 14 AVACS 19, SFB/TR 14 AVACS, 2007. `http://www.avacs.org`.

[FSB06] B. Finkbeiner, S. Schewe, and M. Brill. Automatic synthesis of assumptions for compositional model checking. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE 2006*, volume 4229, pages 143–158. Springer, Heidelberg, 2006.

[FVVC06] O. Florescu, J. Voeten, M. Verhoef, and H. Corporaal. Reusing real-time systems design experience. In *FDL 2006*, pages 375–381. ECSI, 2006.

[Gen35] G. Gentzen. Untersuchungen über das logisches Schließen. *Mathematische Zeitschrift*, 1:176–210, 1935.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[Gie03] H. Giese. A formal calculus for the compositional pattern-based design of correct real-time systems. Technical Report tr-ri-03-240, Lehrstuhl für Softwaretechnik, Universität Paderborn, 2003.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[GTBF03] H. Giese, M. Tichy, S. Burmester, and S. Flake. Towards the compositional verification of real-time UML designs. In *ESEC/FSE-11*, pages 38–47. ACM, New York, NY, 2003.

[Har79] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1979.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of philosophical logic*, chapter II.10, pages 497–604. Reidel, 1984.

[HBB+05] D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, J. S. G. Schellhorn and, and K. Stenzel. The user interface of the kiv verifcation system: A system description. In *UITP 2005*, 2005.

[He89] J. He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[Hei99] S. T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Technical University of Denmark, 1999.

[HHF+94] J. He, C. Hoare, M. Fränzle, M. Müller-Olm, E.-R. Olderog, M. Schenke, M. Hansen, A. Ravn, and H. Rischel. Provably correct systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *FTRTFT 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 288–335. Springer, Heidelberg, 1994.

[HHRS86] R. Hähnle, M. Heisel, W. Reif, and W. Stephan. An interactive verification system based on dynamic logic. In J. H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 306–315. Springer, Heidelberg, 1986.

[HKM07] J. Helin, P. Kellomäki, and T. Mikkonen. Patterns of collective behaviour in ocsid. In T. Taibi, editor, *Design patterns formalization techniques*, chapter IV. IGI Publishing, 2007.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HM05a] J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *FM 2005*, volume 3582 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2005.

[HM05b] J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. Reports of SFB/TR 14 AVACS 5, SFB/TR 14 AVACS, 2005. `http://www.avacs.org`.

[HO02a] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes, data and time. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer, Heidelberg, 2002.

[HO02b] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9:301–334, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Hob07] U. Hobelmann. Verifying properties of processes, data, and time: Linking counterexamples to high-level specifications. Master's thesis, University of Oldenburg, 2007.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, 2006.

[HRS87] M. Heisel, W. Reif, and W. Stephan. Program verification using dynamic logic. In E. Börger, H. K. Büning, and M. M. Richter, editors, *CSL 1987*, volume 329 of *Lecture Notes in Computer Science*, pages 102–117. Springer, Heidelberg, 1987.

[ISO02] ISO/IEC 13568:2002. *Information technology – Z formal specification notation – Syntax, type system and semantics*. International Organization for Standardization, Geneva, 2002.

[ISS09] C. Ihlemann and V. Sofronie-Stokkermans. System description: H-PILoT. In *CADE 2009*, pages 131–139. Springer, Heidelberg, 2009.

[Jon81] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, 1981.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JSS06] S. Jacobs and V. Sofronie-Stokkermans. Applications of hierarchical reasoning in the verification of complex systems. In *Proceedings of the Fourth International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, pages 15–26, 2006.

[KC05] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE 2005*, pages 372–381. ACM, 2005.

[Kna28] B. Knaster. Un théorèm sur les fonctions d'ensembles. *Annales Societatis Mathematicae Polonae*, 6:133–134, 1928.

[KRS07] J. Knudsen, A. P. Ravn, and A. Skou. Design verification patterns. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 399–413. Springer, Heidelberg, 2007.

[KRSS05] V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. Verification of JCSP programs. In J. F. Broenink, H. W. Roebbers, J. P. E. Sunter, P. H. Welch, and D. C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 203–218. IOS Press, 2005.

[KV98] O. Kupferman and M. Y. Vardi. Modular model checking. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 381–401. Springer, Heidelberg, 1998.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Liu09] Y. Liu. *Model Checking Concurrent and Real-time Systems: the PAT Approach.* PhD thesis, National University of Singapore, 2009.

[LLQ+10] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In K. Ueda, editor, *APLAS 2010*, volume 6461 of *Lecture Notes in Computer Science*, pages 1–15. Springer, Heidelberg, 2010.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LX91] K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[MCF+97] Z. Manna, M. Colón, B. Finkbeiner, H. Sipma, and T. E. Uribe. Abstraction and modular verification of infinite-state reactive systems. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering*, volume 1526 of *Lecture Notes in Computer Science*, pages 273–292. Springer, Heidelberg, 1997.

[MD98] B. P. Mahony and J. S. Dong. Blending Object-Z and timed CSP: An introduction to TCOZ. In *ICSE 1998*, pages 95–104, 1998.

[Met10] B. Metzler. *Decomposition for Compositional Verification*. PhD thesis, University of Paderborn, 2010.

[Mey05] R. Meyer. Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. Master's thesis, University of Oldenburg, 2005.

[MFHR08] R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko. Model checking Duration Calculus: A practical approach. *Formal Aspects of Computing*, 20(4–5):481–505, 2008.

[MFR06] R. Meyer, J. Faber, and A. Rybalchenko. Model checking Duration Calculus: A practical approach. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC 2006*, volume 4281 of *Lecture Notes in Computer Science*, pages 332–346. Springer, Heidelberg, 2006.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1980.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[Möl02] M. O. Möller. *Structure and Hierarchy in Real-Time Systems*. PhD thesis, University of Aarhus, Denmark, 2002.

[MORW08] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with UML and Java. *Formal Aspects of Computing*, 20:161–204, 2008.

[MU05] P. Malik and M. Utting. CZT: A framework for Z tools. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 315–352. Springer, Heidelberg, 2005.

[MWW08] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. In S. Liu, T. S. E. Maibaum, and K. Araki, editors, *ICFEM 2008*, volume 5256 of *Lecture Notes in Computer Science*, pages 105–125. Springer, Heidelberg, 2008.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2002.

[OB97] W. R. Oliveira and R. S. M. Barros. The real numbers in Z. In D. J. Duke and A. Evans, editors, *Proceedings of the 2nd BCS-FACS Northern Formal Methods*, Electronic Workshops in Computing, 1997.

[OD08] E.-R. Olderog and H. Dierks. *Real-Time Systems — Formal Specification and Automatic Verification*. Cambridge University Press, 2008.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[OMG09] Object Management Group. *OMG Unified Modeling Language (OMG UML)*. OMG, `http://www.omg.com/uml/`, 2009.

[OS10] E.-R. Olderog and M. Swaminathan. Layered composition for timed automata. In K. Chatterjee and T. A. Henzinger, editors, *FORMATS 2010*, volume 6246 of *Lecture Notes in Computer Science*, pages 228–242. Springer, Heidelberg, 2010.

[Pan02] P. K. Pandya. Interval duration logic: Expressiveness and decidability. *Electronic Notes in Theoretical Computer Science*, 65(6):1–19, 2002.

[PC08] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. In A. Gupta and S. Malik, editors, *CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 176–189. Springer, Heidelberg, 2008.

[Pet09] M. Peters. Erweiterung von Syspect um einen Editor für Phasen-Event-Automaten. Seminar paper, University of Oldenburg, 2009.

[Pet10] M. Peters. Verifikationsarchitekturen in Syspect. Master's thesis, University of Oldenburg, 2010.

[Pla07a] A. Platzer. Differential dynamic logic for verifying parametric hybrid systems. In N. Olivetti, editor, *TABLEAUX 2007*, volume 4548 of *Lecture Notes in Computer Science*, pages 216–232. Springer, Heidelberg, 2007.

[Pla07b] A. Platzer. A temporal dynamic logic for verifying hybrid system invariants. In S. Artemov and A. Nerode, editors, *LFCS 2007*, volume 4514 of *Lecture Notes in Computer Science*, pages 457–471. Springer, Heidelberg, 2007.

[Pla08] A. Platzer. *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems.* PhD thesis, University of Oldenburg, 2008.

[Pla10] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics.* Springer, Heidelberg, 2010.

[PQ08] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, Heidelberg, 2008.

[PQ09] A. Platzer and J.-D. Quesel. European train control system: A case study in formal verification. In A. Cavalcanti and K. Breitman, editors, *ICFEM 2009*, volume 5885 of *Lecture Notes in Computer Science*, pages 246–265. Springer, Heidelberg, 2009.

[PQR09] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 485–501. Springer, Heidelberg, 2009.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *SFCS 1990*, volume II, pages 746–757. IEEE Computer Society, Washington, DC, 1990.

[PR07] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In M. Hanus, editor, *PADL 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer, Heidelberg, 2007.

[RAI92] RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall International, 1992.

[Rav94] A. P. Ravn. *Design of embedded real-time computing systems.* PhD thesis, Technical University of Denmark, 1994.

[Rei95] W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 339–368. Springer, Heidelberg, 1995.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, Reading, 1999.

[Rös94] S. Rössig. *A Transformational Approach to the Design of Communicating Systems.* PhD thesis, University of Oldenburg, 1994.

[Ros98] A. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science. Prentice Hall Europe, 1998.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In L. Kott, editor, *ICALP 1986*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer, Heidelberg, 1986.

[RW03] H. Rasch and H. Wehrheim. Checking consistency in uml diagrams: Classes and state machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS 2003*, volume 2884 of *Lecture Notes in Computer Science*, pages 229–243. Springer, Heidelberg, 2003.

[Sca98] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP.* PhD thesis, University of Oxford, 1998.

[Sch95] S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, 1st edition, 1999.

[Sch08] S. Schewe. *Synthesis of distributed systems.* PhD thesis, Universität des Saarlandes, 2008.

[SH99] G. Smith and I. J. Hayes. Towards real-time Object-Z. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM 1999*, pages 49–65. Springer, Heidelberg, 1999.

[SH02] G. Smith and I. J. Hayes. An introduction to real-time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.

[SH04] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *ICSE 2004*, pages 666–675. IEEE Computer Society, 2004.

[SLD08] J. Sun, Y. Liu, and J. S. Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In T. Margaria and B. Steffen, editors, *ISoLA 2008*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, Heidelberg, 2008.

[SLDC09] J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating specification and programs for system modeling and verification. In W.-N. Chin and S. Qin, editors, *TASE 2009*, pages 127 – 135. IEEE Computer Society, 2009.

[SLDP09] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *CAV 2009*, volume 5643 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2009.

[Smi92] G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.

[Smi00] G. Smith. *The Object Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[SS94] J. U. Skakkebæk and N. Shankar. Towards a duration calculus proof assistant in PVS. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *FTRTFT 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679. Springer, Heidelberg, 1994.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2, Patterns for Concurrent and Networked Objects. John Wiley and Sons, New York, 2000.

[Sti88] C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[STW10] S. Schneider, H. Treharne, and H. Wehrheim. A CSP approach to control in Event-B. In D. Méry and S. Merz, editors, *IFM 2010*, volume 6396 of *Lecture Notes in Computer Science*, pages 260–274. Springer, Heidelberg, 2010.

[Süh99] C. Sühl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM 1999*, pages 29–48. Springer, Heidelberg, 1999.

[Süh02] C. Sühl. An overview of the integrated formalism RT-Z. *Formal Aspects of Computing*, 13(2):94–110, 2002.

[Sys06] Syspect project group. Endbericht. University of Oldenburg, 2006.

[Tai07] T. Taibi, editor. *Design patterns formalization techniques*. IGI Publishing, Hershey, New York, 2007.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Toy98] I. Toyn. Innovations in the notation of standard Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM 1998*, volume 1493 of *Lecture Notes in Computer Science*, pages 193–213. Springer, Heidelberg, 1998.

[TS00] H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 188–208. Springer, Heidelberg, 2000.

[TZY+03] W.-T. Tsai, F. Zhu, L. Yu, R. A. Paul, and C. Fan. Verification patterns for rapid embedded system verification. In H. R. Arabnia and L. T. Yang, editors, *Embedded Systems and Applications*, pages 310–316. CSREA Press, 2003.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS 1986*, pages 332–344. IEEE Computer Society, 1986.

[WC01] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, 2001.

[Weh02] H. Wehrheim. Behavioural subtyping in object-oriented specification formalisms. Habilitation, University of Oldenburg, 2002.

[XM98] Q. Xu and S. Mohalik. Compositional reasoning using the assumption-commitment paradigm. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 565–583. Springer, Heidelberg, 1998.

[ZH04] C. Zhou and M. R. Hansen. *Duration Calculus.* Springer, Heidelberg, 2004.

[ZHR91] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[Zwi91] J. Zwiers. Layering and action refinement for timed systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 687–723. Springer, Heidelberg, 1991.

Sequent Rules

$\Box$-introduction . . . 79
all left . . . 71
all right . . . 71
and left . . . 71
and right . . . 71
assumption$_\Box$ . . . 76
assumption$_\Diamond$ . . . 77
assumption axiom . . . 76
axiom . . . 70
box assumption$_\delta$ . . . 76
box choice . . . 72
box loop . . . 77
box loop gen . . . 77
box step . . . 74
box step$_\delta$ . . . 73
box timed . . . 159
cut . . . 70
dc imply left . . . 160
dc imply right . . . 160
dc noevent . . . 160
dc parallel . . . 160
dc sequence . . . 160
dc sequence noev . . . 160
dc step . . . 160
dc uproc . . . 160
diamond assumption$_\delta$ . . . 76
diamond choice . . . 72
diamond step . . . 74
diamond step$_\delta$ . . . 74
exists left . . . 71
exists right . . . 71
implication left . . . 71
implication right . . . 71
independent uproc . . . 143
interleaving$_\Diamond$ . . . 75
negation left . . . 71
negation right . . . 71
or left . . . 71
or right . . . 71
parallel$_\Box$ . . . 75
parallel uproc . . . 134
parallel uproc$_\Box$ . . . 142
parallel uproc axiom . . . 142
prefix$_\Box$ . . . 72
prefix$_\delta$ . . . 72
prefix$_\Diamond$ . . . 72
process call . . . 72
process equivalence . . . 79
process step . . . 79
R-G uproc . . . 143
sequence$_\Box$ . . . 72
sequence$_\delta$ . . . 72
sequence$_\Diamond$ . . . 72
skip$_\Box$ . . . 72
skip$_\delta$ . . . 72
skip$_\Diamond$ . . . 72
sync$_\Diamond$ . . . 75
weakening left . . . 70
weakening right . . . 70

Glossary of Symbols

General

$\mathbb{B}$	Boolean	42
$\bigsqcap S$	Greatest lower bound of S	88
$\bigsqcup S$	Least upper bound of S	54, 87
$\mathbb{N}$	Natural numbers	45
$\mathbb{R}$	Real numbers	45
$\mathbb{U}$	Universe of all semantical values	47

Semantical domain

$\mathcal{I}_1 \frown \mathcal{I}_2$	Concatenation of interpretations	46
$\mathcal{D}$	Domain of values	34, 42
$\bowtie$	Indicates that a variable represents a parameter of a channel	44
$\mathcal{I}$	Interpretation	36, 42
$\mathcal{M}$	Model	34, 42
$\mathcal{M}'$	Model for primed symbols	42
$\mathcal{M}[v := d]$	Substituted model	43
Time	Time domain	22, 34

Logic

φ', ψ'	Constraints, where every function symbol is replaced with its primed variant	44
$\langle r, g \rangle_{\mathrm{DC}}$	DC R-G formula	145
φ, ψ	Formulae	42
v, w	General variables	43
$\exists\, \overline{x}, \forall\, \overline{x}$	Quantification over a vector of variables	66
$[\![\cdot]\!]$	Semantics of terms and formulae	42
Const	Set of constants or rigid function symbols	42
Form_Σ	Set of formulae over signature Σ	42
Sort	Set of sorts	42
$\mathit{SysVar}(\varphi)$	Set of system variables in φ	42
SysVar	Set of system variables or non-rigid function symbols	42
Term_Σ	Set of terms over signature Σ	42

Σ	Signature	42
$\overline{x}, \overline{y}$	Vectors of variables	66

Processes

$P \stackrel{a}{\Longrightarrow} \overline{P}$	τ-closure of the single step transition relation $\stackrel{a}{\longrightarrow}$	17
$P \preceq Q$	P simulates Q	109
$\langle \mathcal{M}_0, a_1, \ldots \rangle$	Abbreviation of an interpretation (ignoring the exact time of event occurrences)	46
$alph(P)$	Alphabet of a process P	44
$\bot$, $\top$	Bottom and top sets in a lattice	88
$\bullet$	Constraining event occurrences	44
$unconst(P)$	CSP process without data constraints	45
$\mathtt{Proc}^{(\infty)}_{\setminus A, V}$	Either $\mathtt{Proc}_{\setminus A, V}$ or $\mathtt{Proc}^{\infty}_{\setminus A, V}$	47
$\checkmark$	Event representing termination	16
$\Box$	External choice operator	14, 44
$\oplus$	General CSP operator	52
$\oplus_I$	General CSP operator on interpretation level	53
$\setminus$	Hiding	14
$\vert\vert\vert$	Interleaving operator	14, 44
τ	Internal (invisible) event	16
$\sqcap$	Internal choice operator	14, 44
$P_1 \equiv P_2$	Interpretation-equivalence of Processes P_1 and P_2	47
$\Longrightarrow^*$	Multi-step transition with arbitrary events	17
$P \stackrel{w}{\Longrightarrow} \overline{P}$	Multi-step variant of $\stackrel{a}{\Longrightarrow}$	17
$\Vert$	Parallel operator	14, 44
$\rightarrow$	Prefixing	14, 44
P, Q	Processes	44
X, Y, Z	Process identifiers	14, 44
`Stop`	Process representing a deadlock	14, 44
`Skip`	Process representing termination	14, 44
$\sqsubseteq$	Refinement of CSP processes with data	106
$\sqsubseteq_{inst}$	Refinement of CSP processes with data wrt. a refinement relation *inst*	107
$P[R]$	Renaming	14
$\mathcal{I}_{\vert P}$	Restriction of interpretation $\mathcal{I}$ to symbols in P	148
$\pi \setminus X$	Restriction to events not in X	54
$[\![P]\!]\mathcal{M}$	Semantics of processes	46
$\fatsemi$	Sequential composition	14, 44
Channels	Set of all channels	43
Events	Set of all events	14, 43
$Events^{\tau}$	Set of events including τ	16

$Events^{\checkmark}$	Set of events including ✓	16
$[\![P]\!]^{i}\mathcal{M}$	Set of interfered interpretations for a process P	150
$traces(P)$	Set of traces for process P	17
$\sim$	Simulation relation on processes	109
$\mu\, X \bullet F(X)$	Solution of a recursive process equation	44
$sort(c)$	Sort of channel c	43
$P_1 = P_2$	Syntactical equality of processes P_1 and P_2	47
Ω	Terminated process	16, 131
$\texttt{Proc}_{\backslash A,V}$	Terminating unknown process excluding events from A and not changing variables in V	47
$P_1 \equiv_T P_2$	Trace-equivalence of Processes P_1 and P_2	47
$\mathbb{U}_{Events}$	Universe of all events	47
$\mathbb{U}^{\tau}_{Events}$	Universe of all events including τ	47
$\texttt{Proc}^{\infty}_{\backslash A,V}$	Unknown process that does not terminate	47
op	Wild-card operator for $\|, \|\|\|, \sqcap, \Box$, and $\fatsemi$	45

dCSP

$\Box\delta$	Always δ	68
$[P]\delta$	Box operator: after all runs of P δ holds	66
$[P]\Box\delta$	Box operator: during all runs of P δ holds	66
δ, ϵ	dCSP formula	66
γ	dCSP formula or a path formula $\Box\delta$,$\Diamond\delta$	66
$\langle P\rangle\delta$	Diamond operator: after at least one run of P δ holds	66
$\langle P\rangle\Diamond\delta$	Diamond operator: during at least one run of P eventually δ holds	66
$\Diamond\delta$	Eventually δ	68
$Form_{dCSP}$	Formulae of dCSP	66
$\mathcal{I} \models \gamma$	Interpretation satisfies γ	68
$\Box\delta$,$\Diamond\delta$	Path formulae	66
$\psi^{\overline{v}_0}_{\overline{v}}$	Replacement of variables $\overline{v}$ by fresh variables $\overline{v}_0$ in ψ.	74
$(\![P]\!)\delta$	Represents $[P]\delta$ and $\langle P\rangle\delta$, respectively	66
$[\![\gamma]\!]$	Set of all interpretations satisfying γ	68

Sequent calculus

Δ	Antecedent of a sequent	68
$\vdash_{SC}$	Derivability in sequent calculus	69
$\vdash$	Sequent symbol	68
Ψ, Φ	Sets of formulae	68
Γ	Succedent of a sequent	68

CSP-OZ-DC

Symbol	Description	Page
$\mathcal{I} \models F$	$\mathcal{I}$ satisfies DC formula F	24
$\Box F$	F holds always	24
$\Diamond F$	F holds eventually	24
c	CSP-OZ-DC specification	110
$\mathcal{I}, [b, e] \models F$	DC formula F holds on $[b, e]$	24
f	DC function symbol	22
X	DC observable	22
p	DC predicate symbol	22
ℓ	DC term denoting the length of an interval	24
D_X	Finite domain for DC observable X	22
$inst$	Instantiation schema	107
$\mathcal{I} \models P$	Interpretation of a process	30
$\mathcal{I}[\![\cdot]\!](t)$	Interpretation of DC observables and state assertions	22
$\mathcal{I}[\![\cdot]\!](\mathcal{V}, [b, e])$	Interpretation of DC terms and formulae	23
$\widetilde{f}, \widetilde{p}$	Interpretations of functions and predicate symbols	22
$\Delta(a, b, c)$	List of symbols changed by an operation	21
$\boxminus e$	Non-occurrence of event e on a non-singular interval	25
$\not\updownarrow e$	Non-occurrence of event e on a point interval	25
$\updownarrow e$	Occurrence of event e	25
$x!, y?$	Output and input variables	18, 44
$\lceil\rceil$	Point interval	24
VP	Process representing an abstract protocol with data constraints and unknowns	110
$cod_{\mid P}$	Reduction of specification cod to sub-process P	120
$\sqsubseteq_{inst}$	Refinement of processes by CSP-OZ-DC specifications wrt. a refinement relation $inst$	107
$\mathbb{W}$	Semantical values for non-generic Z expressions	20
$[\![cod]\!]^{\mathcal{E}}$	Semantics of a CSP-OZ-DC specification	31
$[\![\cdot]\!]^{\mathcal{E}}$	Semantics of Z expressions	20
$[\![\cdot]\!]^{\mathcal{D}}$	Semantics of Z paragraphs	20
$[\![\cdot]\!]^{\mathcal{P}}$	Semantics of Z predicates	20
Val	Set of all DC valuations	22
$Intv$	Set of closed intervals	23
Obs	Set of DC observables	22
$[\![F]\!]$	Set of interpretations satisfying DC formula F	24
$Vars$	Set of time-independent DC variables	22
$\bowtie$	Symbol distinguishing operation parameters	31
$\frown$	The DC chop operator	24
`main`	The initial process of a CSP-OZ-DC specification	29
$\mathbb{U}$	Universe of semantical values for Z expressions	20
$\mathcal{V}(x)$	Valuation of time-independent DC variable	22

$\langle\!\mid\ \mid\!\rangle$	Z binding of values to identifiers	18
$\Xi(x)$	Z term expressing that x is not changed: $x' = x$	52

Phase Event Automata

$\mathcal{I} \models \mathcal{A}$	$\mathcal{A}$ has run matching $\mathcal{I}$	36
$\mathcal{A} \models F$	$\mathcal{A}$ implements F	37
$strict(\theta)$	Clock constraint equal to θ except that $\leq$ is replaced by $<$	33
$I(p)$	Clock invariant of a PEA location p	34
P^0	Initial PEA locations	34
$\mathcal{M}(t)$	Model of symbolic time constant t	34
P	PEA locations	34
$PEA_{\texttt{Proc}}(P)$	PEA representing an unknown process	130, 131
$\mathcal{A}$	Phase Event Automaton	34
$R(\mathcal{A})$	Region automaton for $\mathcal{A}$	137
$\mathcal{L}(C)$	Set of clock constraints	33
$Clock$	Set of clocks	33
$ClVal$	Set of clock valuations	34
$\mathcal{L}^c(C)$	Set of convex clock constraints	33
C	Set of PEA clocks	34
A	Set of PEA events	34
V	Set of PEA variables	34
$TimePar$	Set of timing parameters	33
$s(p)$	State invariant of a PEA location p	34
$TA(\mathcal{A})$	Timed automata representation of a PEA $\mathcal{A}$	136

Index

A

Abbreviation definition 18
alph 14, 44
Alphabet 14
Antecedent 68
Automata-theoretic approach 159
AVACS 179
Axiomatic description 18

B

Backward simulation 124
Basic type definition 18
Binding 18, 31
Box operator 66

C

capsule 174
Case
- □-case 73
- δ-case 73
- ◇-case 73

Channel 43
ChannelDecl 29
Characteristic process 59
Chop operator 24
Class diagram 174
Class parameter 28
Clock 33
Clock constraints 33
Communicating Sequential Processes 14
Communication-closed layering 171
Completeness 92
Component diagrams 175
`com` .... *see* Operation schema
Concatenation of interpretations 46
Configuration
- interfered 153
- of a PEA 35
- of the region automaton 137

Continuity 54
Convex clock constraints 33
Counterexample traces 26
CSP 14
CSP-OZ-DC 13
- Semantics 30
- Syntax 28
- VA representation 186

D

DC .... *see* Duration Calculus
- counterexample traces 26
- event model 25
- formulae 24
- R-G formulae 145
- terms 23

dCSP 66
Deadlock 67
Decl(*cod*) 107
Declaration 19
Decorations 18
Derivability 69
Design patterns 2
Diamond operator 66
Directed set 54
Discontinuity of parallel composition 55
DL .... *see* Dynamic Logic
Down-simulation .... *see* Simulation
Duration Calculus 22
Dynamic Logic 66

E

Eclipse 178

eCSP ... 42, 47
Event ... 25
  annotated ... 136

F
Failure ... 48
Failures-divergences semantics ... 15
Formal methods ... 1
Formula$_{DC}$ ... 24
Forward-simulation ... *see* Simulation
Free type definition ... 18
Function symbols ... 42

G
Generalised refinement ... 107
GNF ... *see* Guarded Normal Form
Graphical user interface ... 178
Guarded Normal Form ... 56, 130
GUI ... *see* Graphical user interface

I
Incompleteness ... 92
Inductive process structure ... 45
Infinite-branching processes .. 17, 44, 45
Init ... 21
Init(*cod*) ... 31, 107
`Init` schema ... 21
Instantiation schema ... 107
Interface ... 29, 31
interface ... 174
Interfered interpretations ... 149
Interference ... 128, 140
Interference freedom ... 140
Interference interval ... 149
Interpretation ... 36
  interfered ... 149
  of an observable ... 22
Interpretation-equivalence ... 47

L
Labelled transition system ... 15, 45
Layered composition ... 171
Least upper bound ... 54
Limit of authority ... 198
LOA ... *see* Limit of authority
LTS ... *see* Labelled transition system

M
MA ... *see* Movement authority
Matching
  of interpretations ... 36
  of VA processes ... 114
Message variables ... 43
Model ... 20
Monotonicity ... 54
Movement authority ... 5, 191

N
NAME ... 20
NL ... 28
Non-interference ... 128
Non-rigid ... 42

O
Object-Z ... 17, 20
Observable ... 22
Operation ... 21
Operation schema ... 21
OZ ... *see* Object-Z
OZ_State ... 21

P
Paragraphs ... 31
Parallel composition
  data-asynchronous ... 129
  of PEA ... 37
Parameter ... 42
Partial design ... 102
Path formula ... 66
PEA ... *see* Phase Event Automata
Phase Event Automata ... 33
  without stuttering transitions .. 129
Plug-ins ... 179
Predicate ... 19
PrivateInterface ... 32
Process algebra ... 14
ProcessDeclaration ... 29
ProcessEquation ... 29

`Proc` normal form ... 59
Protocol ... 6
Protocol class ... 188
Protocol state machine ... 174
Protocol structure ... 6

R
R-G ... *see* Rely-Guarantee approach
R-G condition ... 142, 143
RCP ... *see* Rich Client Platforms
Recursion ... 14
Reduction of CSP-OZ-DC ... 120
Refinement
  by CSP-OZ-DC specifications ... 107
  of CSP processes ... 106
Refinement rule ... 110
Region automaton ... 137
Region construction ... 135
Relative completeness ... 92
Rely-Guarantee approach ... 141, 142
Replicated operator ... 44
Rich Client Platform ... 178
Rigid ... 42
Rule schema ... 68
Run of a PEA ... 35

S
Safety-critical systems ... 2
Schema ... 18
SchemaText ... 19
Semantics
  CSP-OZ-DC ... 31
  dCSP formulae ... 67
  processes ... 46
  unknown processes ... 50
Sequent ... 68
Simulation ... 109
Slicing ... 170, 185
Slicing criterion ... 170
Soundness ... 69
Stable-failures semantics ... 15
Stable location ... 49
Standard Widget Toolkit ... 178
State assertions ... 23
StateExpr ... 23, 25
State expressions ... *see* State assertions
State machine ... 174
State schema ... 21
Stereotype ... 174
$strict(\theta)$ ... 33
Structured events ... 43
Stuttering transition ... 34
Stutter invariance ... 37
Substitution ... 43
Succedent ... 68
SWT ... *see* Standard Widget Toolkit
Symbolic clock constant ... 33
Syspect ... 174
System variable ... 34, 42

T
TCS . *see* Transition Constraint System
$Term_{DC}$ ... 23
Terminating model ... 67
Termination ... 67
Test automaton ... 159
Test formula ... 159
Timed automaton ... 136
Timing parameter ... 33
Trace-equivalence ... 47
Traces ... 17
Trace semantics ... 15
Transition Constraint System ... 181
Transition relation ... 16
  τ-step ... 17
  multi-step ... 17

U
UML .. *see* Unified Modelling Language
UML profile ... 174
unconst ... 45
Unified Modelling Language ... 2, 174
UnknownProc ... 29
Unknown process normal form ... 59
$untime(\mathcal{I})$ ... 30, 36, 46
Untimed sequence ... 30

V

VA . . . . *see* Verification Architecture
Validity of a sequent 68
Verification Architecture 6
Visibility semantics 31

W

Wizard 178

X

XML export 179

Z

Z notation 17
Z_Paragraph 19