CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Optimizing Development Processes

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Dipl.-Inform. Ralf Hermann Buschermöhle

17. November 2014

Gutachter:	Prof. Dr. Werner Damm Prof. Dr. Martin Törngren

Datum der Einreichung:	17. November 2014
Datum der Verteidigung:	13. März 2015

Author's address:
Ralf Hermann Buschermöhle
Quellenweg 63a
D-26129 Oldenburg
Germany

E-Mail: ralf.buschermoehle@informatik.uni-oldenburg.de

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Doktorarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Der Inhalt der Arbeit wurde nicht bereits für eine Diplomarbeit- oder ähnliche Prüfungsarbeit verwendet. Die Stellen der Doktorarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Statement

I hereby declare that I have written this doctoral thesis independently and did not use any other than the specified resources. The content was not used in a a diploma thesis or a similar work. Text passages of the thesis taken from other sources in letter or sense of are identified by details of the origin. This also applies to drawings, sketches, pictorial representations, and for sources from the Internet.

Abstract

Today, system engineering companies face the challenge to align their processes optimally to the project goals to fulfill the project requirements. These requirements rise continuously because the systems to be developed become more and more complex. Due to globalization effects, having their origins often in the information technology the cost pressure rises at the same time. However, the information technology is in many cases also the solution to this dilemma by offering new, efficient methods and techniques. But they are not a silver bullet and require the alignment to the project goals and development processes.

Process optimization is done on development process models as planing abstraction. Unfortunately the existing process models lack of guidance because of their inherently high abstraction level. The process models tend to be complex and specific when the abstraction level is low because the notations, methods and techniques used in the development processes are often numerous. In order to get precise guidance a formal process model would be required. Therefore it would be necessary to map all process elements (including their static and (possible) dynamic dimension) onto elements of a formal process modeling language. This is currently, with the existing process models and process modeling languages, not possible because the languages are not suitable.

Additionally, the question rises how processes and their models could be kept in sync and what methods and techniques could be used to analyze the models appropriately. Established statistical methods to (manually) survey and analyze the data will often result in high effort in particular when the process elements have a dynamical dimension that needs to be covered. Unfortunately this is often the case because the system under development will likely have a dynamical dimension that is relevant. Only very few companies use established statistical methods and techniques on a level, which would be necessary to answer at least some process questions appropriately. The problem, that the level of abstraction of a development process element likely changes over the time and is often highly subjective makes the task even more complex. This leads to the problem, that many development effects are not analyzed appropriately and not well understood. Also this lack of understanding results in methods and techniques that are not inline with the process targets and encourages a conservative attitude towards new methods and techniques.

Development processes optimization is key to fulfill the current requirements efficiently. This work presents a formal, yet flexible process modeling language that can be used to capture development processes on arbitrary abstraction levels. This modeling language is complemented by analysis and synthesis methods and tools to optimize process models appropriately and to bind process and process model tight together.

The developed software environment was successfully evaluated with the help of industrial partners, complements the work and shows exemplarily optimization scenarios and their effects. Finally, impacts of previously developed artifacts on the synthesized process model activities are computed that can be used as guidance in preceding process steps.

Zusammenfassung

Unternehmen der Systementwicklung stehen heutzutage vor der Aufgabe ihre Prozesse optimal auf die jeweiligen Projektziele auszurichten, um den Projektanforderungen gerecht zu werden. Die Projektanforderungen steigen kontinuierlich, da die zu entwickelnden Systeme immer komplexer werden. Gleichzeitig steigt der Kostendruck aufgrund von Globalisierungseffekten, die oftmals ihre Ursache in der Informationstechnik besitzen. Gleichermassen bietet jedoch gerade die Informationstechnik neue, effiziente Methoden und Techniken zur Lösung dieses Dilemmas. Die Methoden und Techniken sind allerdings kein Allheilmittel, sondern müssen immer wieder auf die Projektziele und damit auf die Prozesse ausgerichtet werden.

Die Optimierung der Prozesse geschieht dabei auf den Planungsabstraktionen der Entwicklungsprozesse, der Entwicklungsprozessmodellebene. Leider helfen die existierenden Entwicklungsprozessmodelle, aufgrund ihres inhärent hohen Abstraktionsgrades, dabei oft nicht ausreichend. Verringert man jedoch den Abstraktionsgrad werden die Modelle schnell umfangreich und speziell, da die in der Entwicklung zum Einsatz kommenden Notationen, Methoden und Werkzeuge zahlreich sind. Zudem müssten die Modelle formalisiert werden, damit sie entsprechend analysierbar sind und präzise Antworten liefern können. Dies würde jedoch bedingen, dass die Prozesselemente - wenn nötig - formal statisch wie dynamisch in dem Prozessmodell abgebildet werden können. Die notwendige Formalität wie auch Flexibilität und Angemessenheit bringen aktuell allerdings weder die Prozessmodelle noch entsprechende Prozessmodellsprachen mit.

Zudem stellt sich die Frage wie Prozesse und ihre Modelle synchron gehalten werden können, und mit welchen Methoden und Techniken selbige zu analysieren sind. Gängige Methoden der Datenerhebung und statistische Methoden zur Analyse sind oftmals nur mit großem Aufwand einsetzbar, insbesondere wenn die Prozesselemente eine inhärente Dynamik aufweisen. Dies ist allerdings häufig der Fall, da das zu entwickelnde System oftmals eine Dynamik hat, die es zu beachten gilt. Nur sehr wenige Unternehmen setzen diese statistischen Methoden und Techniken in einem Maß ein, das notwendig ist, um Antworten auf zumindest einige Prozessfragen zu erhalten. Hinzu kommt, dass die jeweils relevanten Abstraktionsgrade der Entwicklungsprozessmodellelemente über die Entwicklungszeit variieren und oftmals subjektiv sind. Dies führt dazu, dass viele Effekte in der Entwicklung nicht ausreichend analysiert werden können und damit nicht ausreichend verstanden werden. Dieses Nichtverständnis resultiert darin, dass die Entwicklungsmethoden und -techniken nicht optimal auf die Projektziele ausgerichtet werden können und fördert zudem eine eher konservative Einstellung, was den Einsatz neuer Methoden und Techniken angeht.

Die Optimierung der Entwicklungsprozesse ist der Schlüssel, um den aktuellen Anforderungen effizient begegnen zu können. Dazu wird in dieser Arbeit eine formale und gleichzeitig flexible Prozessmodellierungssprache entwickelt und vorgestellt, die in der Lage ist, Entwicklungsprozesse in beliebigen Abstraktionen wider zu spiegeln. Zudem wurden entsprechende Analyse- und Synthesemethoden implementiert, um die Optimierung der Prozessmodelle zu unterstützen und Prozesse und Prozessmodelle eng aneinander zu binden. Die erfolgreiche Evaluation der entwickelten Softwareumgebung geschah mit Hilfe von Industriepartnern und komplementiert die Arbeit und zeigt exemplarisch Optimierungsszenarien und ihre Effekte. Zudem wurden Einflüsse zuvor erstellter Artefakte auf das synthetisierte Prozessmodell berechnet, die zur Orientierung in den vorgelagerten Prozessschritten genutzt werden können.

Acknowledgements

First I want to thank my adviser Prof. Dr. Werner Damm. He always supported my "stubborness" with his time, ideas, and absolute support. The enthusiasm he has for his research was always contagious and motivational for me.

I like to thank the members of the OFFIS group transportation and the conjoint University department. They have been a source of friendships as well as good advice and collaboration. Special thanks to Dr. Bernd Westphal for critical insights and motivation and to Joerg Oelerink who developed some first RMOF prototypes as student research assistant.

The evaluation would have been impossible without Dr. Tom Bienmueller of BTC-Embedded Systems AG who connected RMOF to the real world of an industrial context. I like to thank Prof. Dr. Achim Rettberg to support the development with code generators.

Regarding the synthesis I need to thank Joerg Lehners, who is the perfect instance of an administrator and it possible to "Hit it with iron". Additionally in this context I need to thank Dr. Oliver Melchert who made it possible to compute on the computer cluster of the physics department.

Special thanks go to Prof. Dr. Liane Haak and Dr. Daniel Suepke who reviewed all the writings and presentations and were always a source of inspiration and friendship.

Least no last I'd like to thank my girlfriend Hiemke Schmidt, friends and family for their support and understanding in particular in difficult times.
Thank you. One long journey ends here.

Ralf Buschermöhle

Oldenburg, March 2015

Contents

1 Introduction

> "We measured ten times more errors being caught with our verification framework, compared to the traditional approach (70 vs 7). We also used ten times fewer people (2 instead of 20)."
>
> *Model Checking - Grundlagen und Praxiserfahrungen [1]*
> Gerard Holzmann

All system engineering companies try to optimize their development process in terms of goals like time, costs and product quality. Influence factors of such goals are often numerous and include, e.g., tasks, resource usages, and products as well as the way tools are embedded in the development process. Development process optimization is based on an understanding between influence factors and goals. The degree of the potential influence on a process goal is often proportionally related to the level of objectivity, detail and precision that is required to adequately define and verify the related hypothesis. To verify a hypothesis, appropriate data has to be surveyed, aggregated and analyzed. Optimizations can be introduced into the development process based on the results of the analysis. This is often an ongoing process because either the weights of the process goals or their influence factors change over time.

Traditional methods to model and analyze hypotheses have inherent restrictions. This is reflected in the methods and techniques companies use to plan and optimize development steps and in the gap between planing and reality (see [2], subsections 3.5-3.9). This applies in particular for complex, behavior-related hypotheses that are often required to describe software-driven activities and artifacts. A behavior-related hypothesis describes, e.g., the interactions required to reach a certain test coverage of the system tested. The required test interactions depend on, e.g., complexity of the models describing the system, expertise of the tester and the tool environment that is used to manage and specify the test cases.

Today, many activities of development processes are software driven - because they are often more flexible, more powerful and more efficient than traditional engineering

methods. Concerning the test process this is often represented in different abstractions of the system under test ranging from model in the loop (system and environment are simulated as models), over software in the loop (software for the target platform is generated and simulated on a virtual processor of the target platform), to processor in the loop (generated code runs on a processor of the target platform). The last step in the test process is often a hardware in the loop test. The more components are represented in hardware the higher the precision of the results (if the system is implemented using custom hardware we face often less failures, e.g., due to deviations between software generation and hardware). But the more components are represented in software the lower the costs and development time - in particular when changes need to be done. Today, software is often key to influence the way systems are developed.

For this same reason, software often has a high impact on development process goals. Influence factors include for example a significant but on the same time abstract and minimal description of the software's input and output artifacts. These descriptions often change in their syntax and semantics when the system is developed over the time. Traditional methods for optimizing development processes often lack the required flexibility, power and efficiency compared to the software driving the process. Many companies presumably flinch from applying these traditional optimization methods because of the high effort required to use them. Less than 2% of the approx. 400 companies investigated in the SUCCESS [3, 4] study used standard maturity models to improve their processes based on standardized statistical methods. If the methods are not applied rigorously the relationships between goals and influence factors are often not well understood (see [2], planning horizon of three month with a mean deviation of 20%).

The results of process optimization activities must be reflected in the process model to guide the development activities. Industrial development process models are often informal as in the case of the V-Model XT [5], especially in regard to behavioral aspects. The informality often increases their acceptance, yet makes it impossible to use them precisely or analyze them appropriately to find potential optimizations. Many formal process modeling languages exist, but they are not applied widely (see section 1.4.1). Since most of these languages are Turing complete, their suitability is problematic - not their expressiveness. Suitability is extremely important in terms of all kinds of modeling issues, especially when it comes to development process models. There are many stakeholders, e.g., managers, developers, testers, with all kinds of views on the process, such as work flow related views, schedule-related views and cost-related views. Some of these views change over time, e.g., a function-related view of a developer changes in its abstraction level over the time the system is developed. All views should interpret its process model consistently. A low degree of suitability of a process modeling language makes it difficult to interpret process related documents. In combination with

the (often very high) effort needed to capture the required hypotheses data, conduct appropriate analyses, interpret their results correctly and support the processes with appropriate methods and tools it comes as no surprise that only a very limited number of companies utilize their full potential in optimizing their development processes (see [2]).

All existing process modeling languages are not widely accepted because they lack in terms of unambiguousness, suitability and sustainability. Suitability and sustainability in general can be achieved by using a meta-language approach because each language layer introduces a kind of independence and flexibility to its derived child layers. On the other hand a meta-language restricts the syntax and semantics used to interpret derived child layer languages. The same idea has been used by the Object Management Group (OMG) to define the so called Meta Object Facility (MOF), which is the base for several modeling languages, including the Unified Modeling Language 2 (UML2), the Common Warehouse Meta-Model (CWM), and the Software Process Engineering Meta-Model (SPEM). All these specifications have in common that they are informal.

This thesis introduces a formal and complete variant of MOF, called Rich Meta Object Facility (or RMOF in short), which is used to derive all kinds of formal MOF based languages in a flexible way. RMOF is in particular extended (with respect to MOF) in terms of behavior introducing a set of concurrent State Machines, method operation calls and an action language. This core language is used to introduce semantically different SPEM variants covering different aspects of a development process but can be used for all kinds of languages required to capture different aspects of the process.

Since software-driven activities have often tremendous optimization potentials and behavior-related hypotheses might be difficult to describe, the adequate hypothesis verification results often in a high effort. This thesis focuses on the reduction of this effort. The reduction will be realized by using precise and yet flexible languages derived from RMOF to describe all hypothesis elements like products, activities and tools appropriately. The appropriateness addresses the different viewpoints of different stakeholders, e.g., designer, tester, manager including adequate - for example time dependent - abstraction levels. The methodology includes the (semi-)automatic tracking of all software-driven activities to build a detailed data base of artifacts and interactions. The introduced modeling languages and the methodology to define, refine and abstract model instances of these languages is complemented by optimization methods, which will make it possible to synthesize and analyze RMOF (meta-)models.

In the following section, a screenplay will introduce the potentials of this approach that allows the derivation of scenarios covering the thesis structure.

1.1 Screenplay and Scenarios

Maya is a development process improvement specialist. She is contracted by a large automotive system integrator to identify optimization potential in the company's verification activities. In its recent projects, the company continuously required more and more effort to achieve the specified verification goals. These were indirectly driven by customers requesting new "x-by-wire" functions, e.g., traffic sign and obstacle recognition, car platooning, and automatic parking. All of these functions require very low failure rates for their certification. Failure detection is done during the verification activities. Since these are increasingly turning into a bottleneck for the tight time-to-market schedules, management would like to have their efficiency improved.

To begin with, Maya downloads the relevant parts of the company's development process repository. This repository stores all kinds of tracked process information including all kinds of artifacts, developer interactions and tools used in development projects. All process stakeholders are connected to this repository with corresponding views of the process, e.g., Gantt charts for the verification team manager visualizing the development progress and upcoming activities, work flow views for members of the verification crew and hardware utilization views for administrators. Maya synthesizes a process model based on common activity patterns of the past development projects. The process model has a coverage of 91% of all relevant projects stored in the companies process repository.

In the first analysis, Maya computes the probability of errors that induce deep process iterations. The analysis reveals the manual implementation work between the output of the code generators of the used modeling frameworks/languages, in particular Matlab Simulink, and the input of the testing environment as a main error source. The required implementation work includes the integration and completion of different code fragments of the code generators composing implementation and test cases. The errors occurred during the certification activity and mainly violated the required statement test case coverage. The required test case coverage specifications were violated in that way, that the implemented test cases did not cover the required test cases, or that syntax (e.g., the certification standards require a certain documentation style at each criticality level) or semantics (e.g., pointers are forbidden to manage complex data structures in components with a criticality level greater than 5) were violated. Additional analyses show that relevant certification documents varied from project to project, in one project for certain components and even for a certain component within a project, e.g., due to the outcome of previously conducted risk assessment activities. Since all activities can be conducted arbitrarily in a project, the outcome of previous activities often changes in a project, inducing additional work in all dependent activities to achieve the certification goals. This is in particular the case when the process

iterations are deep. Maya therefore reconfigures the available code generation services to avoid manual implementation work when possible. If it is not possible to avoid manual implementation work, she adds additional checks that notify the developer when relevant certification rules would be violated in the ensuing process steps. The results of these checks are presented in additional views in the development environment, guiding on an adequate abstraction level through implementation, testing and certification documents. She synthesizes several model abstractions for the verification management to support the planing as well as the control phases with a significance level of 63%. These models require only little input and are computed to guide the development in the early stages regarding the required testing effort. Their significance is determined by analyzing the process model after adding and filtering the activities of the process model previously synthesized.

Next, an additional analysis reveals that the required time to conduct certain verification activities depends significantly on the involved people. Maya refines her hypothesis by adding the tool interaction protocols. With an additional analysis she is able to classify the development team members into groups of different skill levels in terms of tool handling. Based on this refined hypothesis, the significance of the impacts on the process model increases to 72%. A lack of highly qualified employees, who are required to conduct certain verification activities often leads to bottlenecks. Based on this knowledge, she refines and optimizes the control flow of the currently running development processes. Maya talks to the vendors developing the tools used to conduct these verification activities and develops some abstractions, which can be used by employees with a lower skill level if the input fulfills certain (syntax and behavior-related) properties and process activities requiring a low certification level. She develops successive on-the-job training supported by the tools in order to gradually increase the skill level of the employees.

Maya assumes that the complexity of the input models is a relevant input to increase the significance level of the impact process model further. In this case the model complexity depends in particular on the model dynamics. Therefore Maya computes meta-semantics for all relevant models in relation to the testing effort based on the complexity assumption and is able to increase the significance to 81%.

Finally, another analysis reveals that in cases where the complexity and size of the input models are very high, the new verification goals of the certification agencies cannot be reached efficiently by the current tool and method repertoire even if there were enough highly qualified employees and no manual implementation work had to be done. In addition, certification goals are becoming even more demanding in all projects. Maya speaks with the corresponding verification tool vendors and pinpoints a possible solution. There is a new tool that is very effective in solving several verifi-

cation tasks, yet there are also some potential drawbacks. For example the employees need to understand the complex tool handling and the tool requires much more computational power than traditional tools. The computing time of the tool in particular depends heavily on certain behavioral aspects of the input. Maya synthesizes a set of patterns to identify tool inputs requiring presumably a high computation time. Based on assumptions such as the learning curve of the employees, Maya calculates and optimizes successively the possible impact ranges of the tool's introduction. The analysis reveals that after an introductory phase of several months, the tool would drastically reduce the required verification time and would help to meet the certification goals efficiently in the following years with a significant probability.

Based on the analysis' results, Maya meets with all executives from the development division to discuss the process optimization. After this meeting she refines her assumptions and computes the analyses again. She analyzes potential integration strategies for all of the changes. Then Maya arranges a meeting with the management of the automotive system integrator company and presents her results. Management votes for a conservative integration strategy for some development projects currently running at a high priority level, and for a less conservative strategy for the remaining projects. Maya introduces the process changes and updates the company's project database in order to track the additional data. She continuously checks the data in order to react to unpredictable situations. Some minor changes occur but after six months, management decides to implement the process changes throughout the entire company.

The optimization approaches of the previous screenplay lead to the following three scenarios:

(i) Development process (meta-)modeling is required as rich hypotheses based on addressing the question as to what is required to understand the hypotheses correctly, e.g., activities, artifacts and tools. This includes in particular syntax and semantics of all captured process elements. In addition, views are required presenting the appropriate language in different perspectives, e.g., schedule, budget or quality related aspects.

(ii) Development Process Methodology is used to track and enact on hypothetical data directly from the software environment of all process stakeholders. Abstractions and refinements are used to map between different levels of languages and their abstractions in the process, e.g., a concrete and complete implementation of a component vs. all relevant time-related aspects of this component.

(iii) Methods and techniques to analyze and synthesize development process models. Synthesis is used to generate (parts of) a process model that describes some real world phenomena like a formula expressing mathematical relationships between

the input of an activity and the required time to do that activity. Analysis is used to check process model properties, e.g., whether or not a milestone can be reached or to determine the significance of hypotheses like that the complexity of the input model changes the behaviour of tester. Optimization targets questions like if there is a (more) optimal process flow and is often a combination of analysis and synthesis.

The structure of the thesis is derived in the next sections starting with a discussion on process modeling.

1.2 Process Modeling

Traditionally, a process modeling language consists of elements like activities, roles and milestones. When the process model acts as a hypothesis frame, it is necessary to adequately express each hypothesis within the boundaries of the process modeling language. Since the process modeling language should be able to describe all kinds of processes these languages/language elements tend to be abstract. This applies in particular when it comes to precise, formal semantics and even more when it comes to dynamics. If, on the other side, the abstraction level of the described elements is low it is inevitable to use a modeling language that matches the process elements as close as possible. The introduction of the thesis structure is started by discussing some (traditional) process modeling elements briefly to sketch a rough language scope. Process elements include:

- Planning-related elements like activities, tasks and milestones can be described statically in form of a set of achievable targets. The achievement of a target is then controlled manually. On a more fine granular level it is also possible to include dynamic-related elements, like available resources in terms of Petri-Net based flows to support a (semi-)automated checking if an activity can be executed. Figure 1.1 shows the mapping of a Software Process Engineering Metamodel (SPEM [6]) based process model onto Place/Transition Petri Net semantics, whereby the SPEM model is depicted on the left side and the Petri Net on the right side. The activities “Component Implementation”, “Component Testing”, and “Component Integration” are places in the Petri Net. An additional place in the Petri Net represents resource synchronization between “Component Testing” and “Component Integration”. There exist seven (hardware) tokens in the Petri Net. The activities “Component Implementation” and “Component Testing” require a single token. The activity “Component Integration” requires two tokens. The Petri Net restricts the execution of the activities with respect to the available tokens. Figure 1.2 shows nearly the same process model but

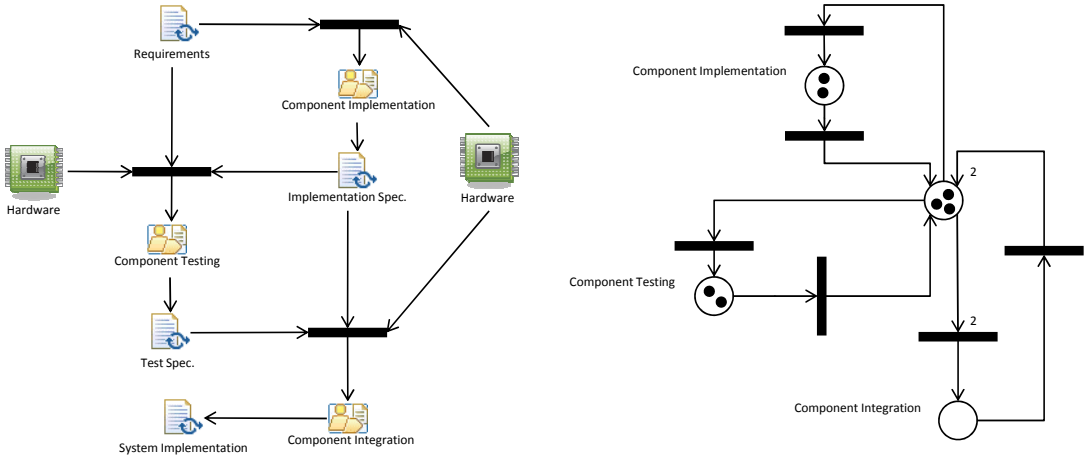


Figure 1.1: Control flow 1: hardware restrictions

with a restriction on human resources, i.e., “Tester” and “Developer”. In contrast to figure 1.1 the semantics of the SPEM model is mapped onto a Colored Petri Net. The places on the Colored Petri Net are the same compared to the Place Transition Petri Net but transition firing requires not only the availability of tokens but also a minimal qualification of the colored token (representing a human resource). The qualification of the human resource token rises after an

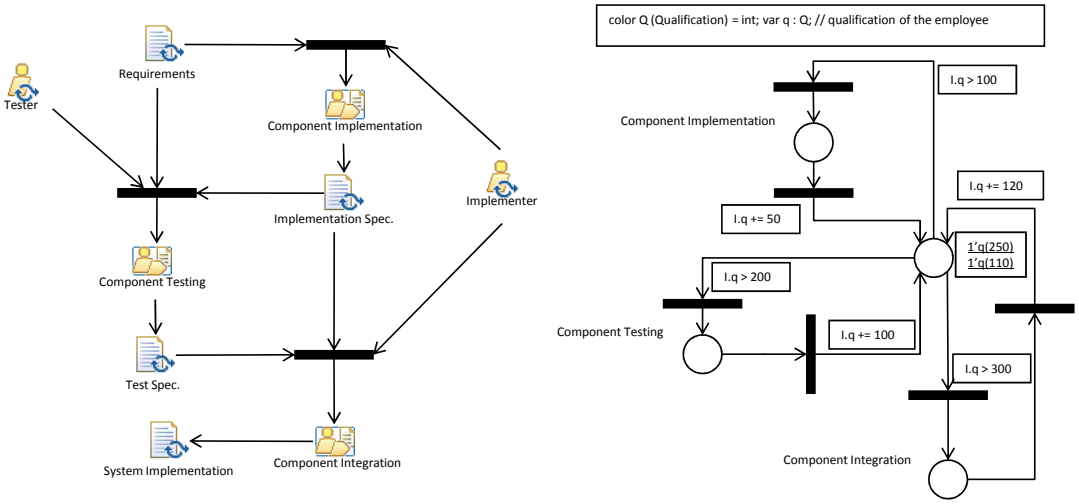


Figure 1.2: Control flow 2: human resource restrictions and qualifications

activity has been conducted. For example the "Component Integration" requires two implementers/testers. One must have a qualification > 250 points, the other one requires a qualification > 110 points. After the activity has been done the qualification of both rises 120 points.

- Resource-related elements are for example hardware, software and humans. Hardware can have a certain number of CPUs, an amount of RAM and a hard disk speed. All these properties can vary over time, e.g., available CPUs in relation to current workload, available RAM in relation of reserved memory and hard disk speed in relation to storage allocation rate. These temporal properties could be described in form of a function or a queuing model. Another example of resource dynamics is time consumption of an algorithm representing the involved software. Figure 1.3 maps the previous example and adds a "Test Software" element which has a computation complexity/required time to process the test cases described in the State Chart on the right side of the figure. The computation depends on

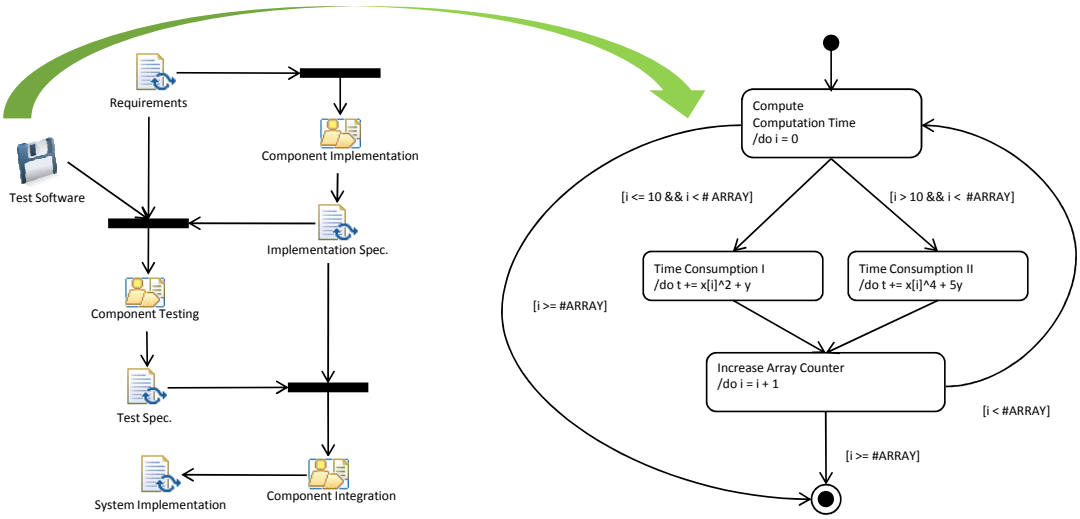


Figure 1.3: Software model: computation complexity of test cases

the number of test vectors available. If there are less than ten test vectors (i represents the number of test cases). the computation time equals $x[i]^2 + y$. If there are more than ten test vectors the computation time equals $x[i]^4 + 5y$. $x[i]$ represents the test vector dependent computation part and y represents the test vector independent computation part.

- Product-related elements are for example system architecture, implementation and test cases. Products can be described statically, e.g., number of methods

of an implementation file or dynamically to compute the achieved test coverage, e.g., state or transition coverage of the implemented unit tests of a certain system component.

- Psychological-related elements are for example motivation, team coherence and social skills. These elements can be described statically, e.g., the motivation of each team member is surveyed at the beginning and at the end of a project. The motivation of a team member could also be described dynamically during the whole project, e.g., based on financial incentives offered or the current team stress level.

- Organizational-related elements are for example roles, number of employees at a company and work constellations. A static description is obvious in this case but roles could also change dynamically with respect to assigned tasks and work constellations could base on the current psychological profile of the team members.

- Analysis-related aspects like probabilism, different degrees of concurrency and interleaving are required to apply analysis techniques. This also includes languages for expressing temporal properties that should be verified, held or optimized during an analysis.

All of these elements should be described with formally defined and suitable languages. The different languages and their semantic elements are combined in the process model semantics. Other potential relevant semantic elements are, e.g., atomicity, concurrency and probabilism. Since every process stakeholder has different tasks to fulfill in a process and to reach (possibly) different (sub-)goals, there exist also different views on the process. Each view requires an adequate set of syntax and semantics on an appropriate abstraction level. Figure 1.4 shows in the middle the previous SPEM model, with refined syntax (left: "Data Structures") and semantics (right: "Algorithms") covering test case data structures and algorithms. On the left side the "Component Testing" activity is refined in terms of different implementation tasks. The screenshot on the lower right side combines the models in the environment of the developer.

There exist many formal process modeling languages, ranging from more general ones, like Petri Nets [7], to specialized ones, like Merlin [8] for human resource management, AND/OR graphs [9] for product/configuration management or SLANG [10] for activity management. For a more complete list see 1.4.1. Since Petri Nets are already Turing complete, later language approaches focus more on the suitability [11], taking aspects like the modeling effort, comprehensibility of the models and aspects besides process control (e.g., product management) into account. Later process modeling languages

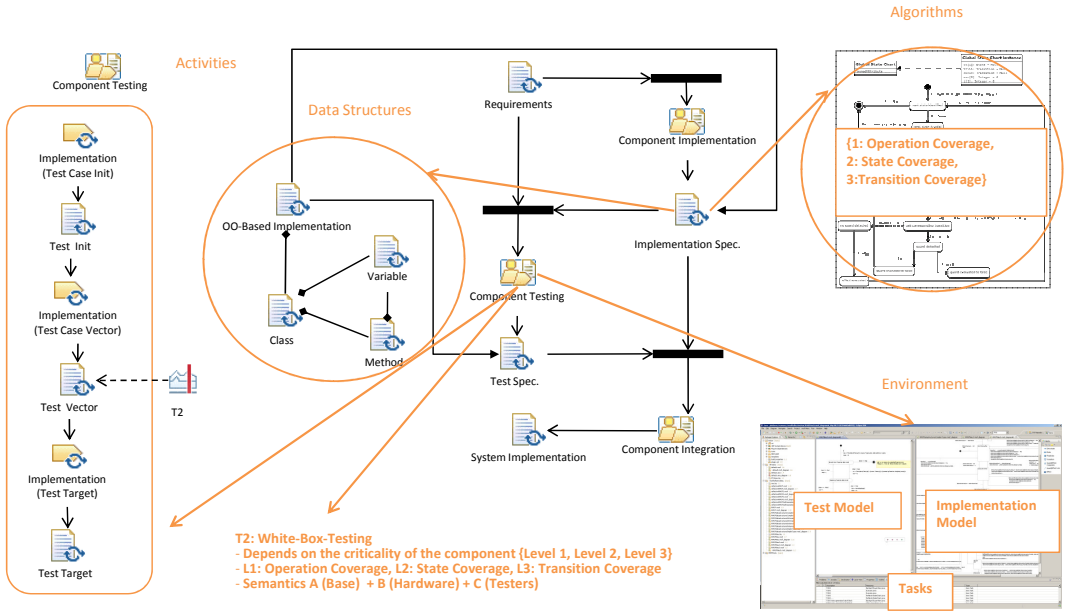


Figure 1.4: Process view examples

try to combine different semantics or distinguish between micro and macro process levels [12].

Nevertheless, no language is suitable in all contexts, simply because the process elements or the process goals vary or, in other words, they are not sustainable. Therefore general language frameworks are required, where it is possible to define arbitrary languages syntactically and semantically. Graph transition systems [13] have a long history in this field, but the suitability is also important regarding the meta-language definition.

The approach introduced in this thesis targets the flexible creation and analysis of process (meta-)modeling languages as instances of a language nucleus including different views for all process stakeholders based on the idea presented in [14] on base of the so called “Rich Meta Object Facility” [15]. This core language is used to embed all kinds of languages and their abstractions including languages to describe processes, artifacts (e.g., source code) or tools. The methodology (complementing the core language) is introduced in the next section and complemented by appropriate synthesis and analysis methods.

1.3 Methodology

The methodology is driven by several aspects in this work. The first aspect focuses on the data acquisition from the conducted development process. The second aspect targets methods and techniques to synthesize models from the tracked data. The third aspect concentrates on the analysis of the models. The (optional) fourth aspect addresses methods to propagate the results directly back to the development process.

Traditionally, data acquisition from a development process is done manually, which has several drawbacks, mainly high resource consumption, errors or inaccuracy and interference with development activities. An automatic approach requires to connect the software used in the process with the process models in a rigorous way, or - in other words - a seamless transition between models and the software environments. This reflects the statement from Leon Osterweil from 1987 that "Software processes are software too" [16].

In order to connect models and software this closely it is required to access the software environment on a very fine granular level regarding available APIs, components or source code itself. Since the effort to connect a software environment this close is high, a software environment covering several development activities would require less effort. Today, there are some open-source, component-based development frameworks available, that address several kinds of development activities. The approach in this thesis targets the enrichment of the Eclipse platform[1], because it is by far the most advanced open source and component based approach[2] that is also widespread among all kinds of developers and offers a rich set of so called plugins for different development tasks[3]. The Eclipse platform was used to develop plugins to track information and to introduce a flexible modeling approach with rich powerful graphical editors.

As mentioned previously, the first aspect of the methodology addresses the automatic surveying of data directly from the software environment of the process stakeholder, including interactions (e.g., mouse button pressed and GUI elements used) as well as modified artifact (e.g., model inputs and test execution results). Figure 1.5 shows a screenshot of the Testvector Editor of the BTC Embedded Systems augmented with the tracking capabilities. Number one in figure 1.5 starts or stops the tracking.

[1] www.eclipse.org
[2] **O**pen **S**tandard **G**ateway **i**nitivative Platform - www.osgi.org
[3] More then 10 Mio. solutions (plugins and features) according to marketplace.eclipse.org, September 2014.

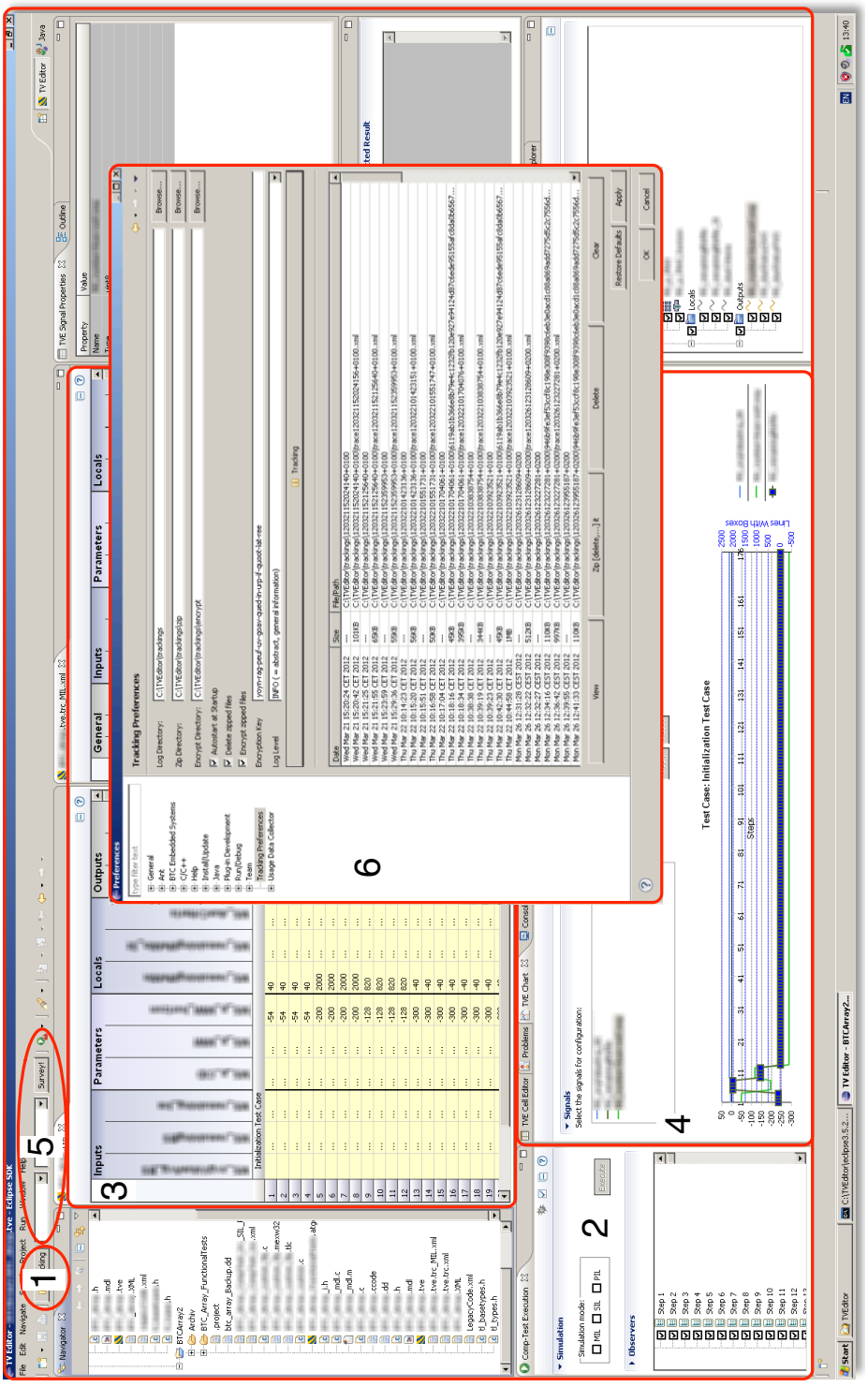


Figure 1.5: Process information tracking

After the tracking has been started (1) several actions of the Eclipse environment (2) are automatically tracked (e.g., activation of plug-ins, switching of perspectives, screen size changes). In particular the matrix to specify test vectors is parsed (3) including the data changed (4) and Matlab Input Models or test execution results. In addition to the automatic tracking the developer can introduce manual data (5). The tracking environment is complemented by preferences to specify where to store log files, to define automatic splitting size, and to set options to encrypt the log files (6). This preference dialog keeps also track of all created log files and can be used to compress and delete transferred files. The tracked data consists mainly of interactions from the developer, environmental settings and artifact data, e.g., the tests edited. The tracking environment is introduced in more detail in section 5.1 on page 160.

Second, the methodology addresses the synthesis of models. The first step of the synthesis is to identify unconditional behavioral patterns. An unconditional behavioral pattern is a set of unconditionally linked (unique) interactions done by the developer. Each interaction consists of all elements to "replay" the interaction, including the GUI element where the interaction took place, the interaction kind (e.g., mouse button, keyboard key and combinations like CTRL + mouse button) and the effects of the interaction (e.g., a change in the test vector matrix, a test execution, a view change). The idea of a behavioral pattern is visualized in figure 1.6. The left side of figure 1.6

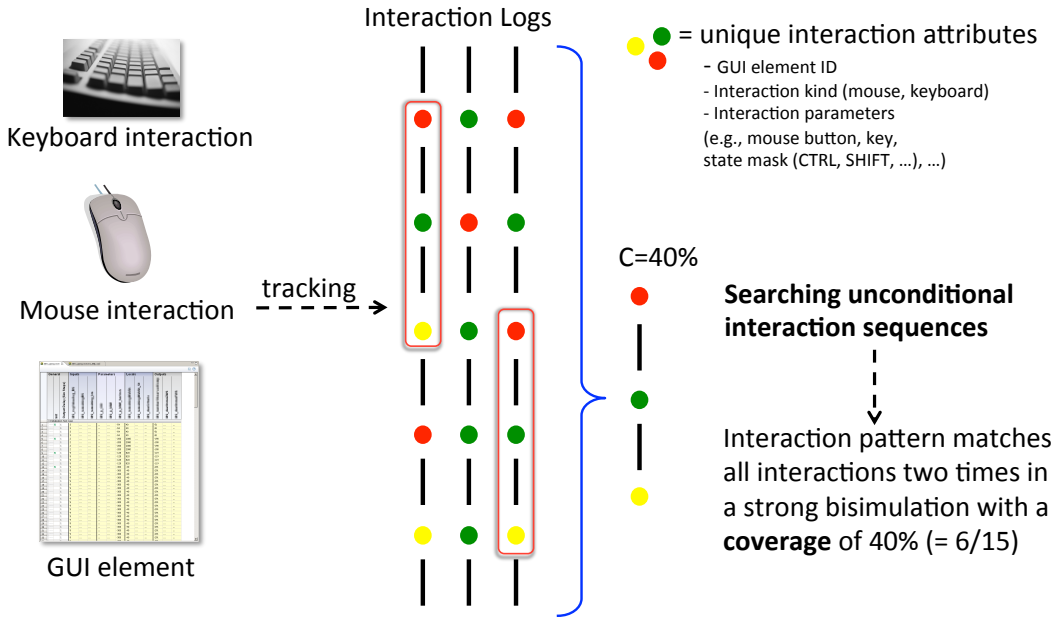


Figure 1.6: Behavioral pattern

displays the two main interaction sources and an example GUI element. The interaction logs are visualized by a set of circles, whereby each circle represents a unique interaction. The identified behavioral pattern on the right side of figure 1.6 presents a sequence of interactions that covers 6 of the 15 interaction sequences tracked in the presented interaction logs (coverage = 40%). Figure 1.7 shows a simple interaction model and the different pattern elements. The top of figure 1.7 is a graphical representation

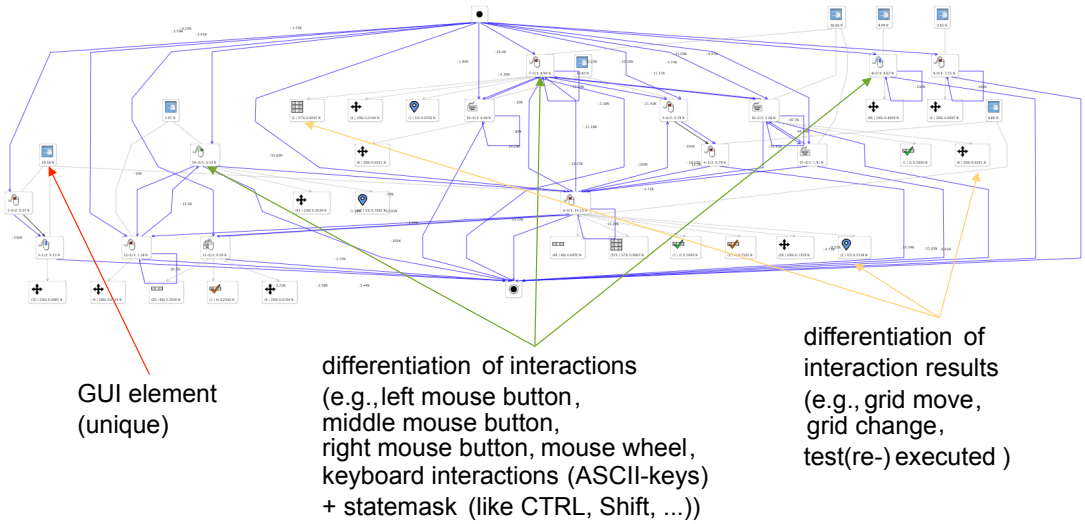


Figure 1.7: Interaction model

of an interaction pattern. Top element is the start interaction. The bottom element is the final interaction. The interaction in-between are differentiated in mouse and keyboard interactions. Each interaction includes different states like pressed modifier keys (e.g., CTRL). Blue transitions with probabilities represent interaction changes. Black transitions represent fix interaction changes (with 100% probability). The higher the transition probability the more interesting the interaction sequence for any kind of optimization. Each interaction has a GUI element assigned where the interaction took place (e.g., the test vector grid). The dashed arrows indicate interaction results, e.g., a test vector grid move, or adding, removing, and changing test vectors, as well as test executions. Test executions and indirectly the interactions that took place before the test execution needed to be differentiated into successful test executions and failures.

A second synthesis step generates conditions to distinguish behavior patterns. The transitions of the models of the first analysis step are enriched by a set of conditions to increase their probabilities if the conditions are evaluated to true. Each condition is based on assumptions influencing the tracked interactions (e.g., complexity of the input model). Figure 1.8 shows an example of this second analysis step. The model

part in yellow shows interactions to implement test vectors. The model part in blue shows interactions to execute tests. The black transitions represent the unconditional transitions of the first model step. The dashed transitions in orange are substituted by the dashed transitions in green in the second analysis step. In this case the cyclomatic complexity of the input model guards the transition t. The results of the synthesis of the evaluation are presented in section 5.4 on page 163 and the following sections including some optimizations. Third, process analysis computes the probability dis-

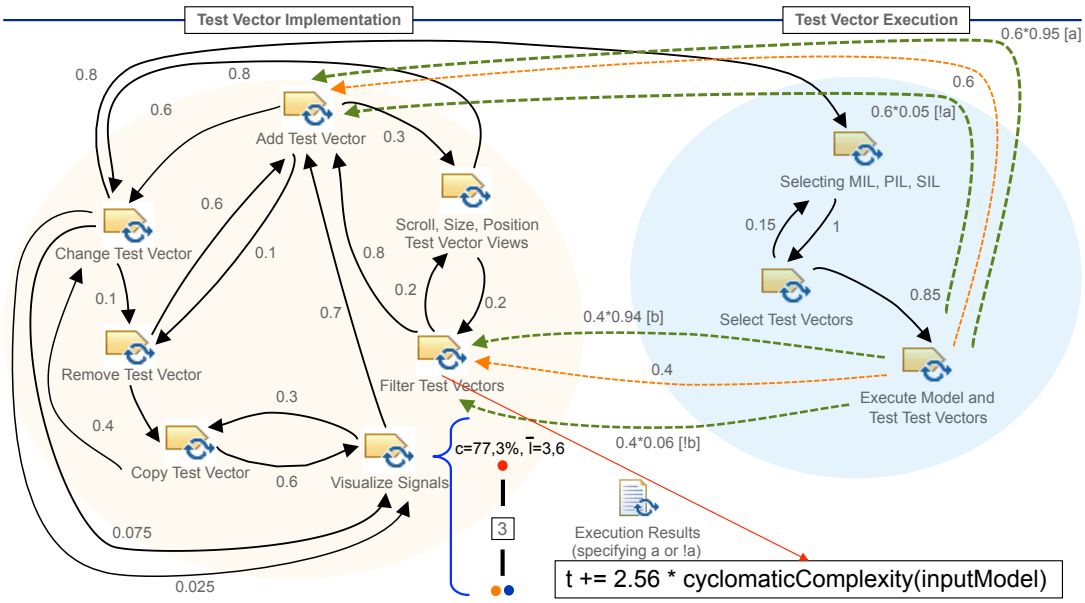


Figure 1.8: Interaction model with hypothesis enrichment

tributions of the interaction effects. This is done by simulating interaction sequences up to a given minimal probability. The results of the analysis of the evaluation are presented in section 5.6 on page 180.

Fourth, process enactment addresses the propagation of the analyzed process model information back into the development environments of each process stakeholder, e.g., active tasks in workflows for the software architect, cumulated costs for the account manager and Pert Diagrams with achievable milestones for the development manager. Figure 1.9 depicts this process. Starting at the upper left corner, there exists a process modeling environment where additional information (e.g., available and assigned resources for each task) can be specified. This modeling environment is connected with the process analysis environment to analyze all kinds of questions (including tracked process informations). The Process Control Center is able to visualize process informa-

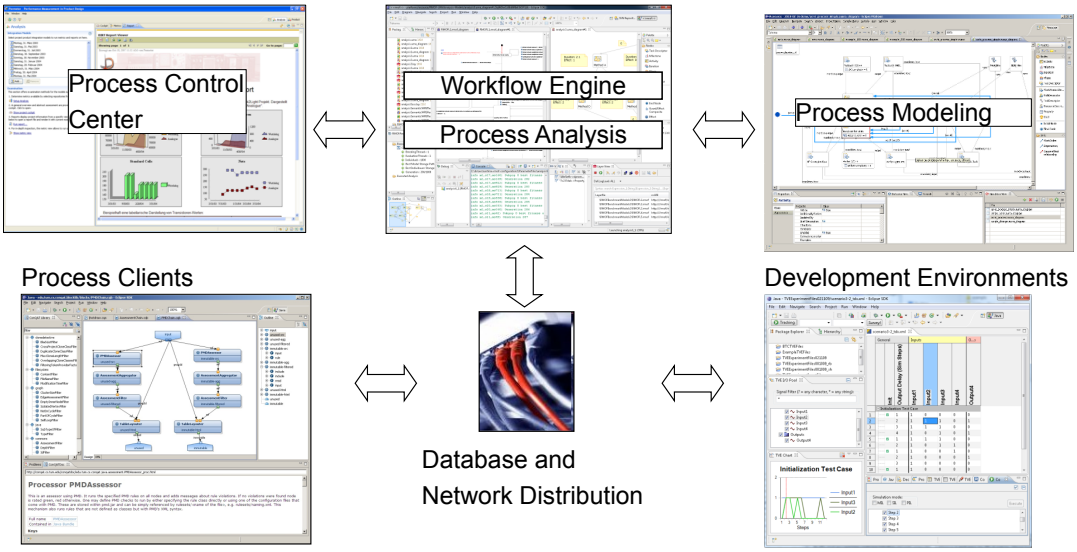


Figure 1.9: Process information distribution

tion (e.g., history of person month occupied by a certain task) in statistical diagrams. The workflow engine executes Process Models. A special network layer distributes all kinds of process information over the network to special Process Clients (e.g., to select the next task) or directly into the development environments (e.g., if design constraints are not fulfilled). The workflow engine was developed in the projects "Speculative and Exploratory Design in Systems Engineering" (SPEEDS[1]) and "Cost-Efficient Methods and Processes for Safety Relevant Embedded Systems" (CESAR[2]) and successfully applied with the industrial partners.

The four elements tracking, synthesis, analysis, and distribution of process information compose the methodology of this work. In combination with the flexibility to create and analyze all kinds of models based on the so called Rich Meta Object Facility, which is described in chapter 2. The next sections introduce exemplary related work starting with some process modeling languages that emerged over the years.

1.4 Problem and Related Work

A hypothesis identification in the context of process model optimization is the task of finding a sustainable relationship between a set of independent effects on a (presum-

[1]http://www.speeds.eu.com
[2]http://www.cesarproject.eu

ably) dependent effect. Figure 1.10 illustrates an example with two effects. In the lower

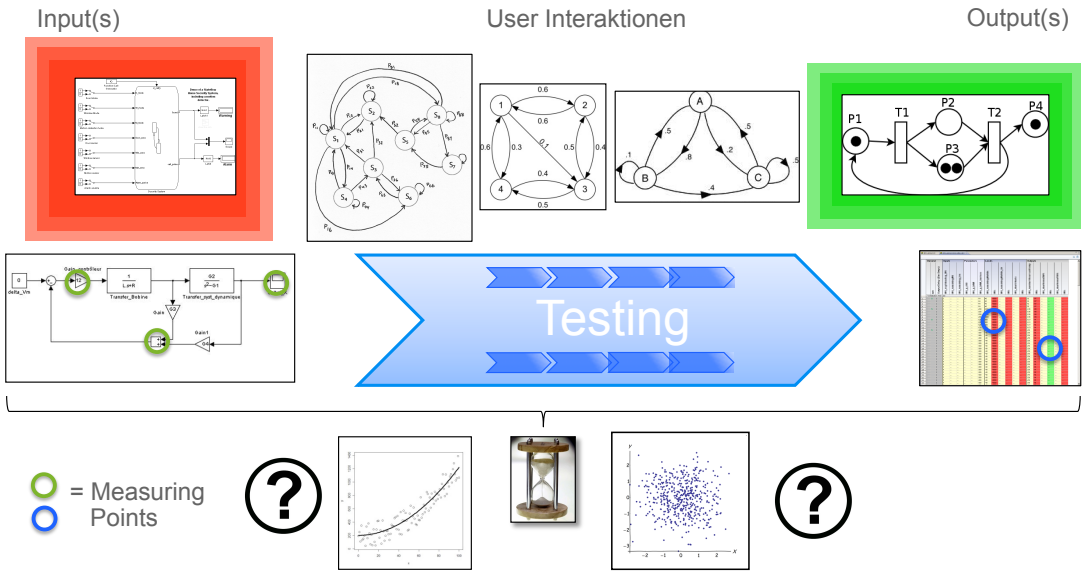


Figure 1.10: Hypothesis verification

part of figure 1.10 are scatter plots depicted that are typically used to sketch surveyed data. A scatter plot is a diagram using Cartesian coordinates to display values for two variables for a set of data. The data is displayed as a collection of points. The horizontal position of each point is determined by one variable, the vertical position of another one. The two scatter plots are approximated by mathematical functions which are widely applied to describe quantitative effects in development processes. Mathematical functions in general are powerful enough to describe all kinds of relationships. But the methods that compute the mathematical functions often have restrictive prerequisites and/or they limit the generated function classes very restrictively, e.g., a multivariate regression analysis requires normal distributed data and calculates only linear mathematical functions. But its not their expressiveness that lacks, its their suitability. For the same reason e.g., C++ is used instead of Assembler to write software and a set of multidimensional, linear functions is often an unsuitable abstraction of the behavior of a software component. The suitability of a language addresses its ability to easily model and interpret language instances with respect to the context of the interpreter. Regarding a human interpreter the interpretation often depends on the number and complexity of language constructs required to construct the model. The suitability is context dependent, e.g., a task in a process context is often function, goal and time dependent. The same applies for the C++ program in the implementation phase. Abstractions of this program are developed in the Design phase (e.g., with

UML diagrams like Class Diagrams, State Charts or Sequence Diagrams).

The second crucial aspect of process models and process modeling languages is their sustainability. The sustainability of a process model and process modeling language is proportional related to the number of observations that can be explained over the time. For example a spline is a smooth piecewise-polynomial function. The method can easily explain a number of dots in a scatter plot. The sustainability is (nearly) zero, because if another dot is introduced it is likely that the spline needs to be recomputed and even worse, the spline does not "explain" anything (assuming that the observations the previously sketched points are based on are not chosen as explication itself). Figure 1.11 shows on the left side three scatter plots without an obvious mathematical relationship because they have a dynamic dimension which is not covered by the mathematical functions. The middle shows a RMOF model defining meta-behavior based on a previous analysis of artifacts and behavior restricting potential behavioral semantics. The right side shows three different model instances explaining the dynamics. The upper one includes probabilism, the middle one is a Matlab Simulink Stateflow model and the last one is a Petri Net based model.

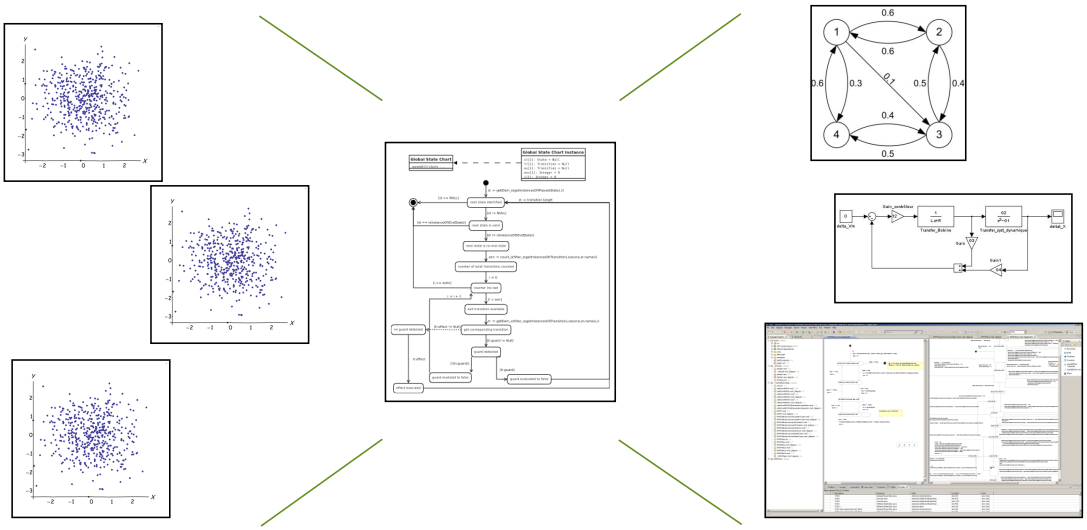


Figure 1.11: Sustainability and suitability

In order to achieve both suitability and sustainability, a certain (controllable) degree of flexibility is required. The next section introduces process modeling languages to describe (different aspects of) process models.

1.4.1 Process Modeling Languages

The idea of process development languages and their execution started with the paper "Software Processes are Software too" by Osterweil [16]. Over the years several process modeling languages have been developed, but none has been widely accepted. If the language is not able to cover the complete process, it is often classified into the categories "Process Specification Languages", "Process Design Languages", and "Process Implementation Languages". The languages range from informal ones to formal ones and ones with an enactment environment[1]. The next enumeration shows developed process modeling languages prefixed by the year of their introduction.

1962 Petri Nets [17] were introduced to model chemical processes and are widely used to define semantics of process modeling languages. Over the time a lot of variants

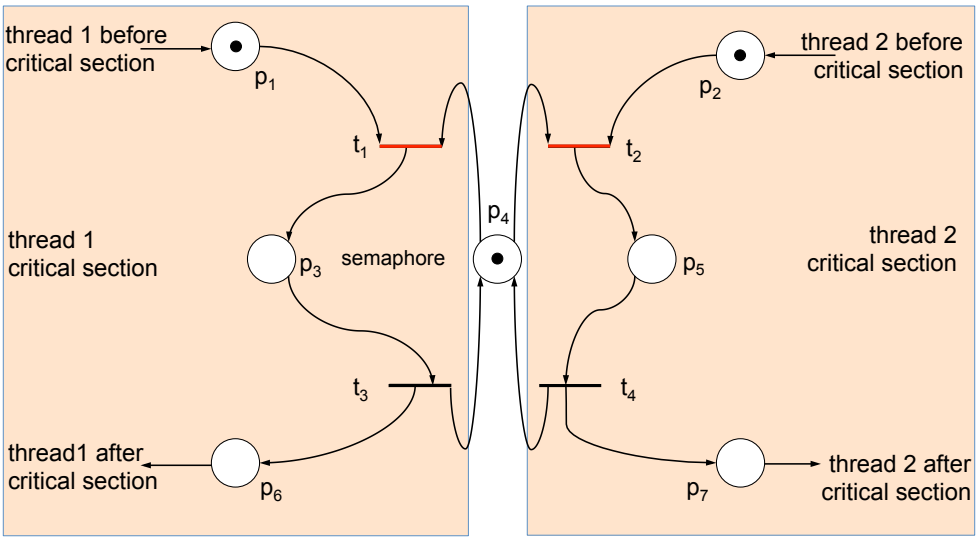


Figure 1.12: Petri Net example with critical section

appeared with a higher abstraction level [7], to express time [18], or self modifications [19]. All variants can be expressed as standard place/transition Petri Net. Figure 1.12 shows a critical section modelled in a Petri Net model.

1981 IDEF0 [20] Functional Modeling method is designed to model the decisions, actions, and activities of an organization or system and was derived from the graphic modeling language Structured Analysis and Design Technique. The notation is informally specified and can be used to describe data flows, system controls, and

[1] The short descriptions rely often heavily on abstracts and chapters referenced by the process modeling language

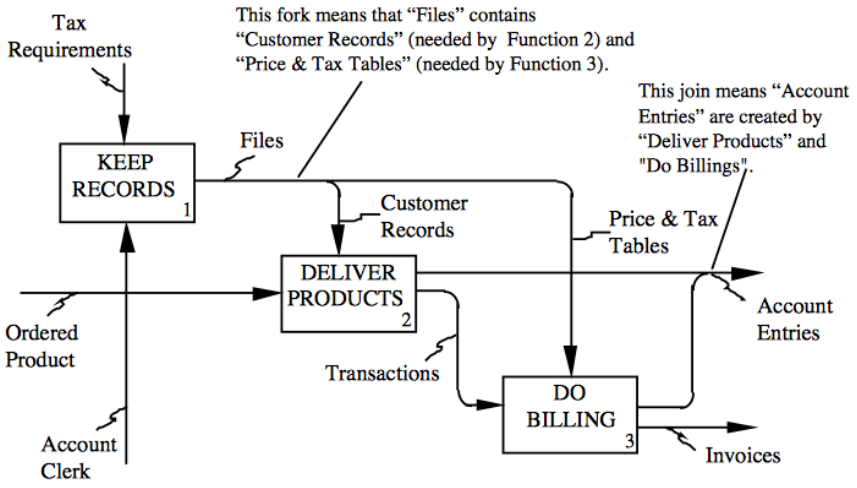


Figure 1.13: IDEF0 example (see [20], page 24)

the functional flow of life cycle processes. An output arrow may provide data or objects to several boxes via the forking mechanism, as shown in figure 1.13.

1988 Grapple [21] defines processes in a hierarchy using so called plan operators with multiple levels of abstraction. Each operator has some precondition defining the

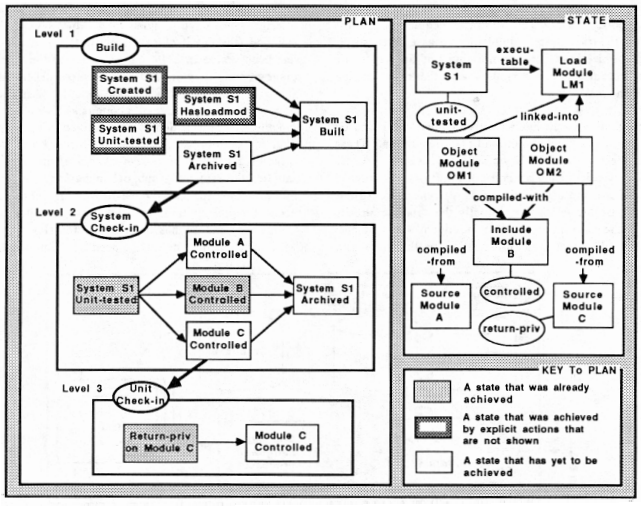


Figure 1.14: Grapple example (see [21], page 102)

state that must hold in order for action to be legal, and a set of effects which

defines the state changes resulting from performing the action. A plan places stress more on goals rather than activities. Grapple combines plan generation with plan recognition. Plan generation automatically executes process steps to achieve a goal while plan recognition attaches steps executed by the process performer to the current set of plans. Figure 1.14 shows a subset of a system development process on the left side and the process states on the right side.

1988 Marvel Strategy Language (MSL) is the process modeling language for the process centered software engineering environment Marvel [22]. The software process model in Marvel is an extensible collection of rules for the process steps with pre-conditions and post conditions. Figure 1.15 shows an example MSL model for a programming team where multiple developers are not permitted to change the same module at the same time but need to reserve a module before changing it. Marvel interprets its rules using forward and backward chaining. Forward chain-

```
not reserved(module) and saved(module)
    { reserve module }
reserved(module, userid);

reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

for all components k such that in(module, component k)
    and uses(component k, component c):
        reserved(module, userid)
    { change component c }
```

Figure 1.15: MSL example (see [22], page 11)

ing is letting Marvel perform opportunistic execution of process steps as soon as their precondition is satisfied as a result of prior steps performed. Backward chaining helps Marvel find the process steps whose post-conditions satisfy the pre-condition of other process steps that have been activated.

1989 In Hierarchical and Functional Software Process (HFSP [23]) software processes are described as a collection of activities, which are characterized by their input and output relationship and defined as mathematical functions. Complex relationships can be decomposed into sub-activities together with the definition of their input and output. Figures 1.16,1.17 show two different ways of representing the same activity composition. The first one of figure 1.16 is suitable for understanding hierarchical structure of activity decomposition. The second figure 1.17 is suitable to identify attribute dependency. The enactment mechanism in HFSP provides activity scheduling, activity execution management, tool

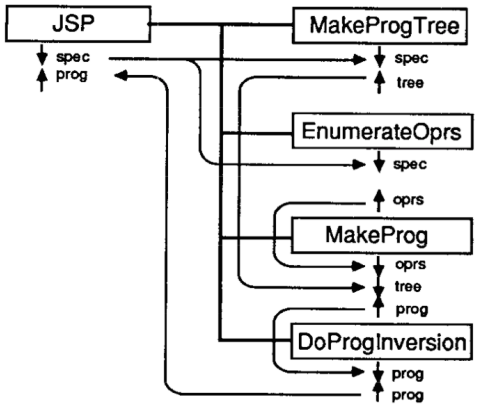


Figure 1.16: HFSP example 1 (see [23], page 346)

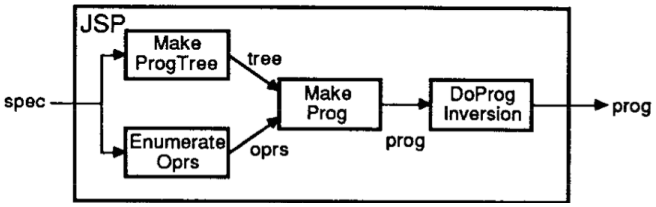


Figure 1.17: HFSP example 2 (see [23], page 346)

invocation, access to input and output of the software process model, and user interaction. Activity scheduling allows concurrent activities to execute when their input becomes available.

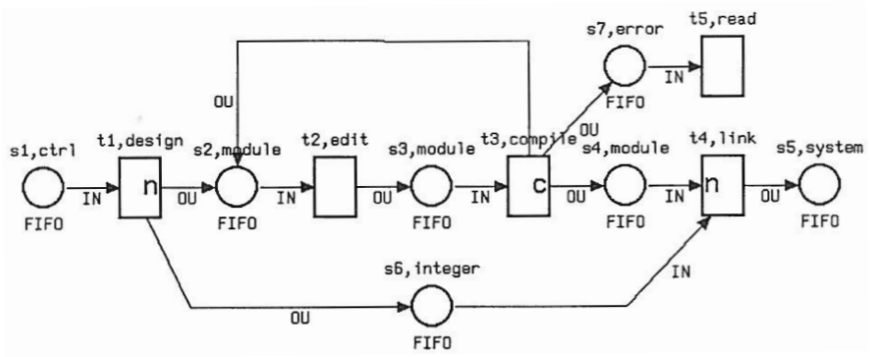


Figure 1.18: MELMAC example (see [24], page 4)

1990 Melmac [24] is the predecessor of the so called FUNSOFT Nets, defined as a Petri Net extension with additional views like Object Type and Activity View, Process View, and Project Management View. Figure 1.18 shows a partial software process model. This model was used to motivate different firing behaviors. t_1 has an object flow based firing behavior (s6 stores the number of identified modules the designed system will contain) whereas t_3 has a so called complex output firing behavior depending on the successful compilation of the module. If a module cannot be compiled it must be re-edited.

```
Example
resource_model Designer(eff_0: Resource_effort) is
    resource_interface
        imports
            resource_attribute_model Resource_effort;
        exports
            effort: Resource_effort := eff_0;
    end resource_interface

    resource_body
        implementation
        -- An instance of this model represents a single member of the design team.
        -- Persons assuming the role of a designer must be qualified.
    end resource_body
end resource_model Designer
```

Figure 1.19: MVPL example (see [25], page 13)

1991 MVP-L [26] is an informal language designed to model processes, products, resources, and quality attributes and support their instantiation in project plans. Figure 1.19 shows a fragment of the resource model 'Designer'. The effort assigned to this resource states the effort available to this resource for executing some process in the context of a project plan.

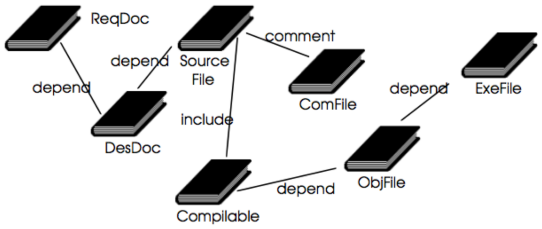


Figure 1.20: E3 example (see [27], page 11)

1993 E3 [27] is an informal, object-oriented software process modeling language consisting of four views: inheritance view, task view, functional decomposition view and informational perspective view. Figure 1.20 describes a user view on the

product configuration schema, relations are differentiated into several categories and read from bottom to top.

1993 EPOS [28] is based on configuration management capturing the evolution of systems. A process is the total set of engineering activities to produce and evolve

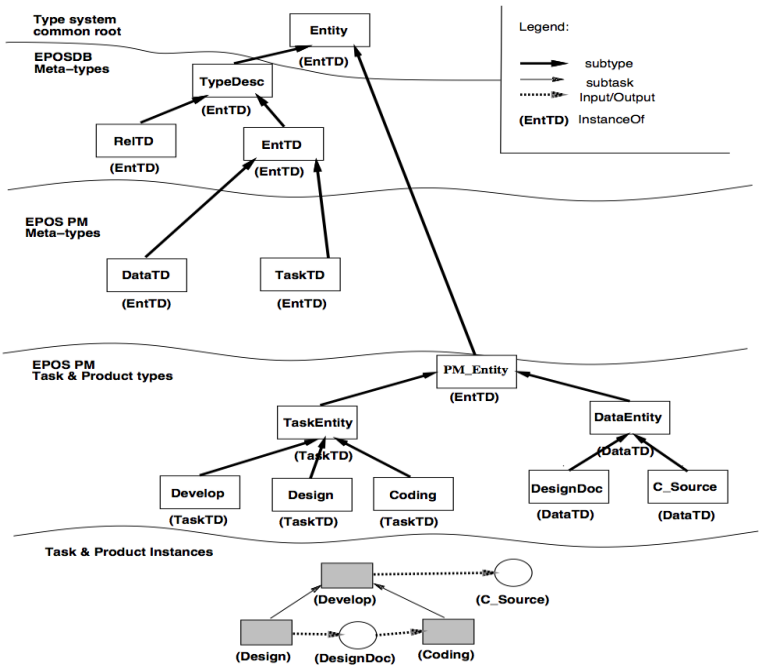


Figure 1.21: EPOS example (see [28], page 5)

these systems. Therefore process modeling and configuration management are strongly connected. EPOS uses configuration management techniques to model the evolution of systems, introduces rule based techniques in the process models, graph/net based techniques for enactment and process programming to express basic activity steps. EPOS supports meta-modeling as visualized in figure 1.21, shows meta-typing, types and instances in EPOS.

1994 Process modeling in Merlin [8] is done in the two levels process design and process enactment. Process design is supported using a graphical notation called Entity Relationship Models and State Charts. State Charts are used to specify the behavior of the process models. Figure 1.22 presents two role specifications of a quality assurance engineer and a programmer and a finite-state machine for the attribute state of the document c-module during an step implementation.

1994 PADM [29] is based on the so called formal “Base Model” (BM [30]). The se-

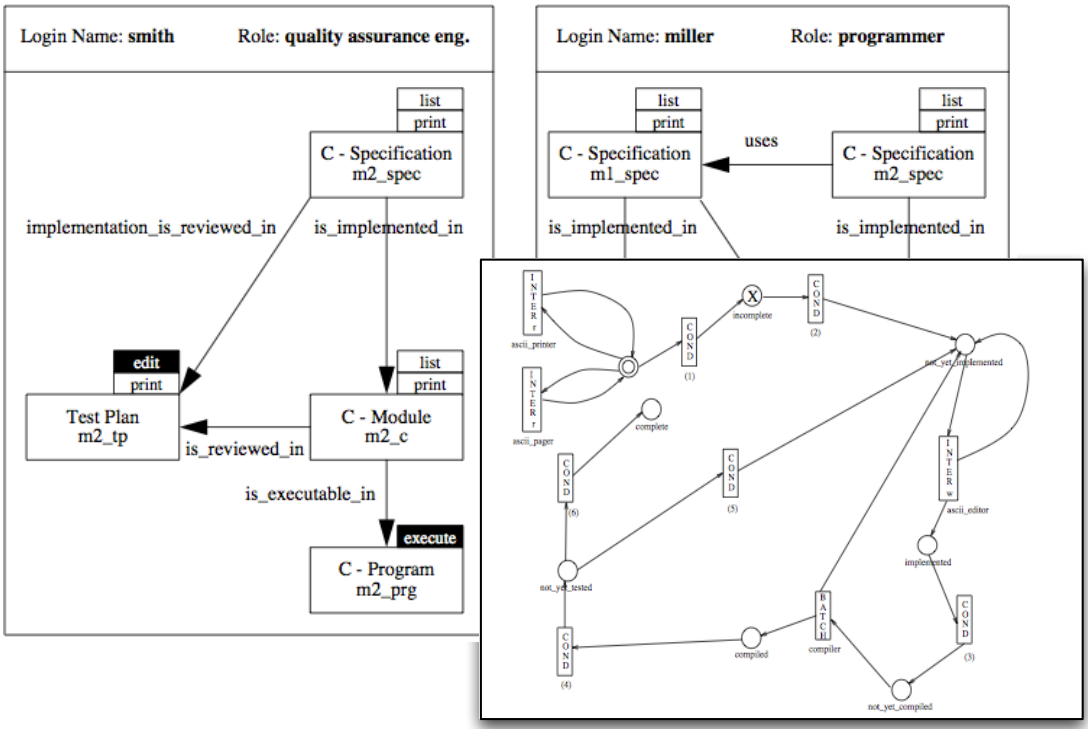


Figure 1.22: Merlin example (see [8], page 5, 14)

mantics of BM constructs are defined using a linear time temporal logic. In BM, a system is considered to be composed of components which may be executed in parallel. Each component provides a number of operations and contains state variables which can only be accessed by the operations. An operation may call other operations in other components. All components of PADM are specified in BM. Figure 1.23 shows a so called Role Activity Diagram defining a role behavior. They specify the different activities (of a person with this role) which can take place and dependencies which exist between them.

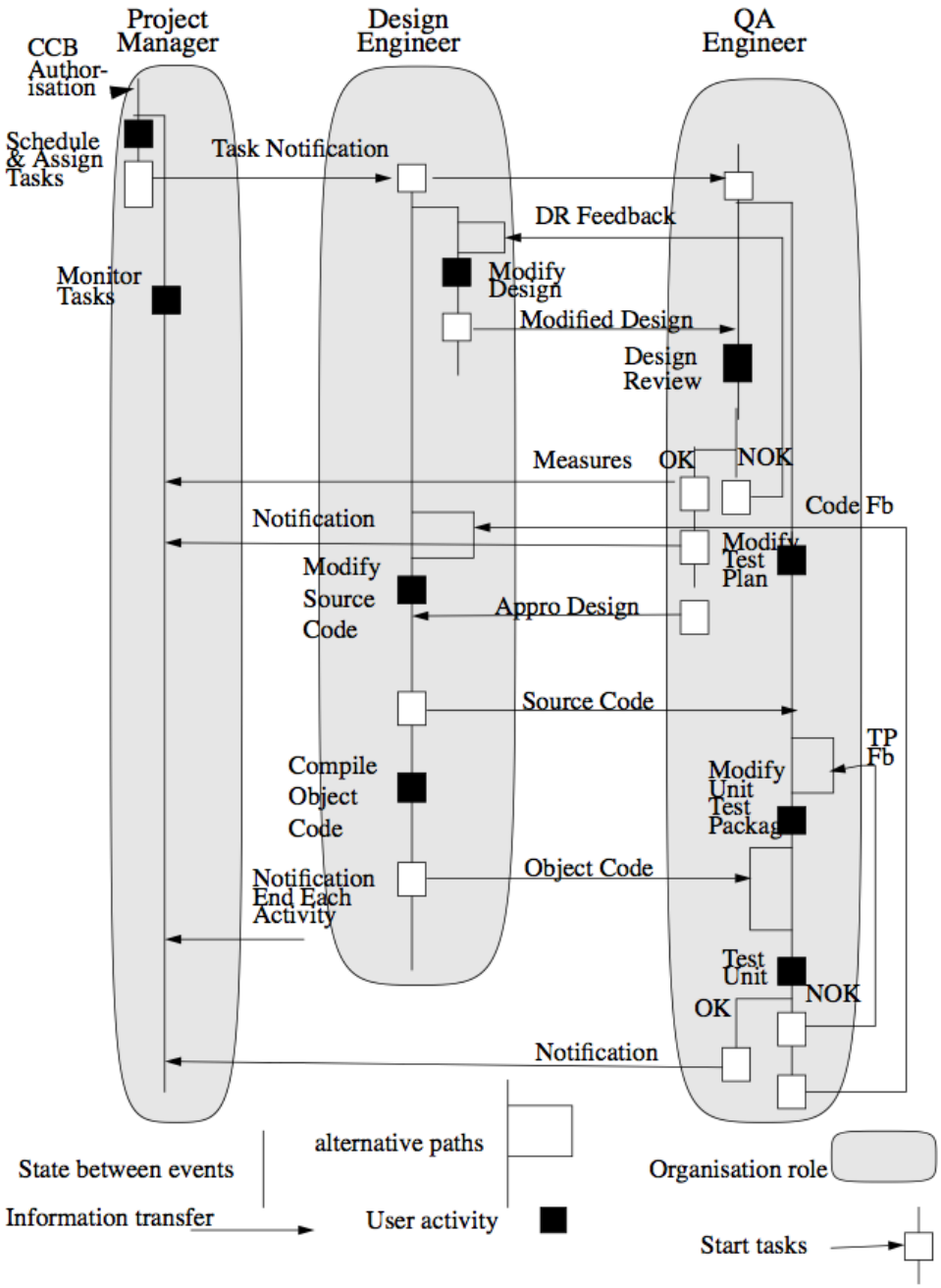


Figure 1.23: PADM example (see [29], page 20)

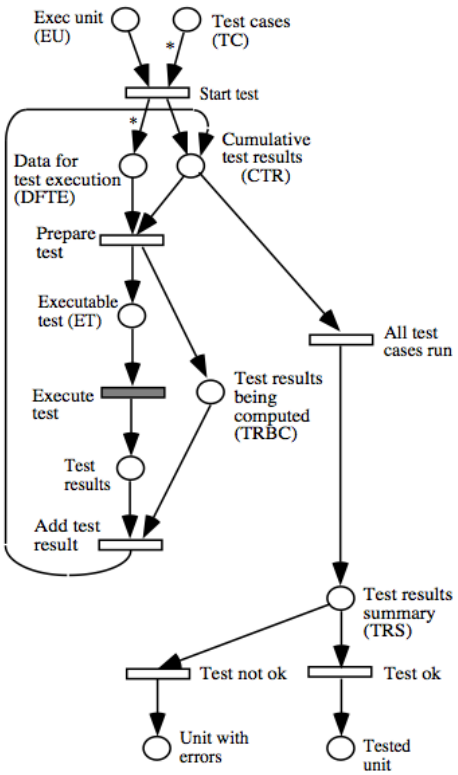
1994 The Model for Assisted Software Process Description Language (MASP/DL) is the process modeling language for the framework ALF to build process-centered software engineering environments [31]. A MASP software process model is com-

```
audit : (IN _p:project; OUT _r:report)
 PRECOND: audit_authorized(_p) = TRUE
 POSTCOND: audit(_p)=TRUE AND proj_to_rep(_p,_r,NO_KEY)
 KIND : INTERACTIVE
```

Figure 1.24: ALF example (see [31], page 21)

posed of software process fragments consisting of an Entity Relationship Attribute to describe data, a set of operator types to allow abstractions of tools and pre/post conditions, a set of rules of type event-condition-action, a set of ordering constraints and characteristics. Figure 1.24 shows an audit action specification.

1994 Slang is a process modeling language for SPADE [10]. Like Melmac, Slang uses




```
event StartTest (EU: ExecUnit, TC: set (TestCase);
                 DFTE: set (ExecTest), CTR: TestSummary)
  local test: TestCase, data: ExecTest;
  guard
    forall test in TC: test->name = EM->name
  action {
    DFTE = set();
    for (test in TC) {
       data = new ExecTest;
       data->name = EU->name;
       data->authorName = EU->authorName;
       data->sourceCode = EU->sourceCode;
       data->compilerUsed = EU->compilerUsed;
       data->execCodePath = EU->execCodePath;
       data->testData = test->testData;
       DFTE = DFTE + set(data);
    }
    CTR->testingUnit = EU->name;
    CTR->results = set();
    CTR->pendingTestsNb = count(TC)
  }
end
-----------------------------------------------
event PrepareTest (DFTE: ExecTest, CTR: TestSummary;
                   ET: ExecTest, TRBC TestSummary)
  guard
    CTR->pendingTestsNb > 0
  action {
    ET = DFTE;
    TRBC->testingUnit = CTR->testingUnit;
    TRBC->results = CTR->results;
    TRBC->pendingTestsNb = CTR->pendingTestsNb - 1
  }
end
-----------------------------------------------
event AllTestCasesRun (CTR: TestSummary; TRS: TestSummary)
  guard
    CTR->pendingTestsNb == 0
  action {
    TRS = CTR
  }
end
```


Figure 1.25: SPADE example (see [10], page 12,13)

a Petri Net extension as its base. A Slang process model can be hierarchically structured as a set of activities, each is described by a net that may include invocation of other (sub) activities. Like in high-level Petri Nets, process data are represented as tokens in Slang. Figure 1.25 presents a test phase model using Kernel Slang on the left side. Additional guards and actions are presented on the right side.

1995 FUNSOFT Nets [32] are high level Petri nets for software process modeling. Figure 1.26 shows a waterfall driven software process modeled with FUNSOFT

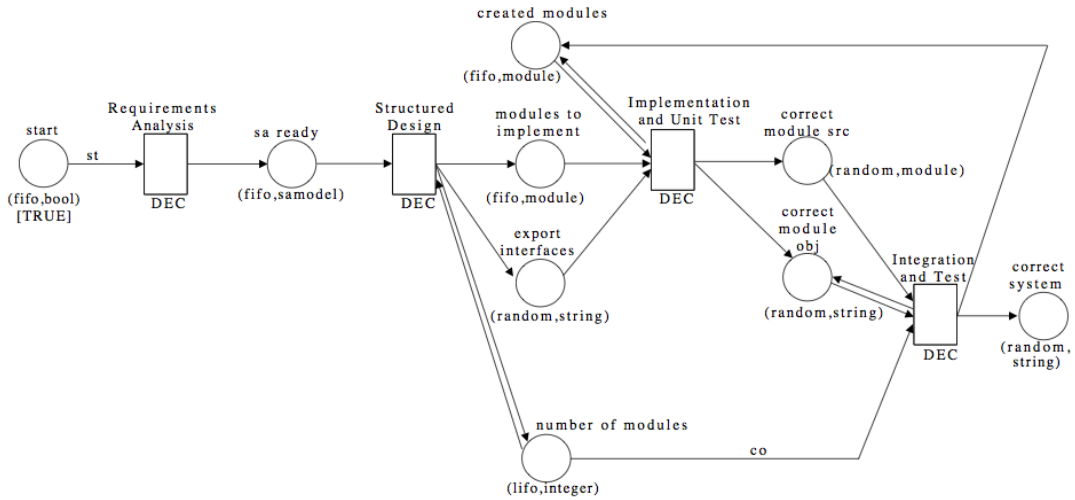


Figure 1.26: FUNSOFT example (see [32], page 9)

nets. Some nodes have additional semantics, e.g., an Decision Node (DEC) manages resources/persons executing the task.

1997 JIL [33] text based process modeling language based on the programming language ADA [34] developed by a team led by Jean Ichbiah of CII Honeywell Bull. JIL differences so called proactive and reactive controls. Pro active controls are executed imperatively, whereby reactive controls react autonomically, e.g., on events like product states. Figure 1.27 shows a reactive control specification describing in the upper part that the object identification in the object analysis and design based on the Booch method should be revisited if the class identification is not completed yet. There exists a subset of JIL called Little-JIL which comes with a graphical editor. Figure 1.28 shows the data flow of a flight reservation process.

1997 The Concurrent Software Process Language (CSPL [36]) shares most of its syntax

```
REACTIONS Identify_Classes_And_Objects_Reactions IS
-- Reactions for Booch Process Step Identify_Classes_And_Objects
BEGIN
    REACT TO COMPLETION OF Identify_Objects BY
        IF NOT Complete(Identify_Classes) THEN
            INVOKE SUBPROCESS Identify_Objects;
        END IF;
    END REACT;

    REACT TO UPDATE OF Reqts_Spec BY
        TERMINATE Identify_Classes_And_Objects;
    END REACT;
END Identify_Classes_And_Objects_Reactions;
```

Figure 1.27: JIL example (see [33], page 8)

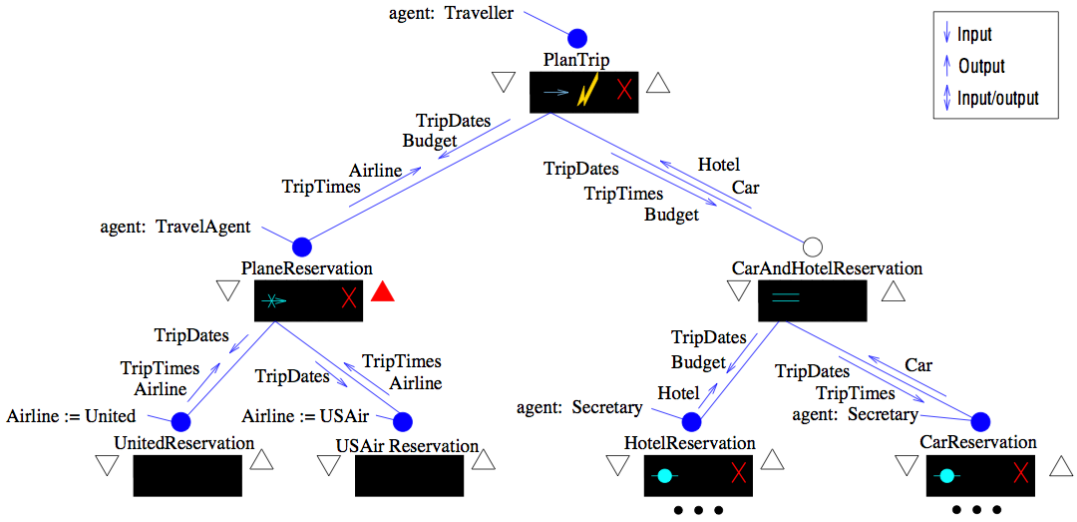


Figure 1.28: Little JIL example (see [35], page 7)

with ADA95 and adds special language constructs to model software processes consisting of work assignment statements, communication-related statements to synchronize tasks, role unit statements to define mappings between roles and developers, tool unit statements to specify tools that are required to complete a task and relation unit work assignment statements allowing the assignments of work units to multiple developers. Figure 1.29 shows the information of the modification of a test plan "ModifyTestPlan" and the modification of the unit

```
    "ModifyTestPlan"
task body ModifyTestPlan is

begin
 loop
 ...
 testplan.modify(test_plan, design_doc);
 inform ModifyUnitTestPackage to set start;
end loop;
end;

    "ModifyUnitTestPackage"
task body ModifyUnitTestPackage is
   start : event;
begin
 loop
  waitfor start;
  testpac.modify (test_unit, test_plan);
  inform TestUnit to set test_available;
 end loop;
end;
```

Figure 1.29: CSPL example (see [36], page 6)

test package "Modify Unit TestPackage".

1997 APEL [37] is a visual, formal process modeling language. In APEL, a software process is described using Object Modeling Technology diagrams, data flows, control flows, workspaces and cooperation, roles, and state transition diagrams. Figure 1.30 shows some concurrent states (with dashed lines) and local state

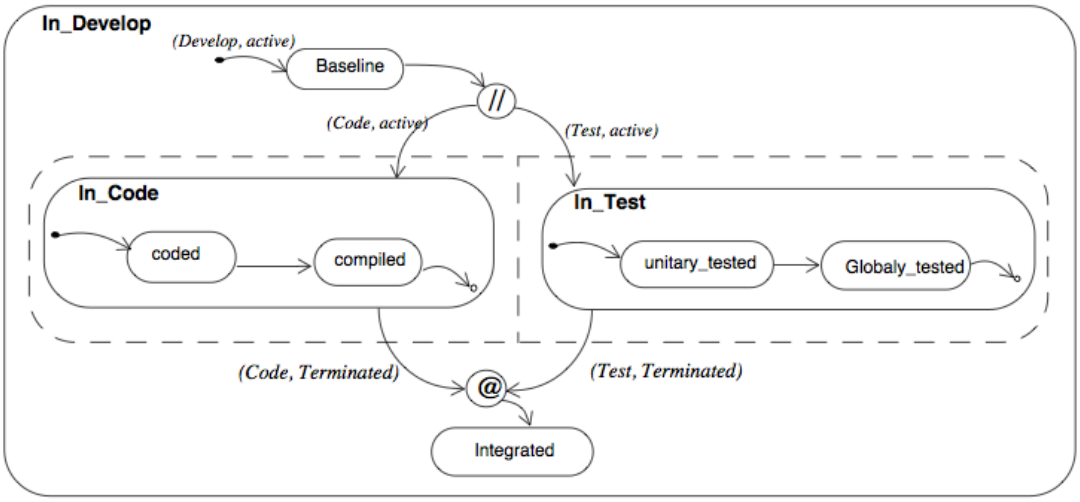


Figure 1.30: APEL example (see [37], page 27)

diagrams. Leaving the dashed area forces a re-synchronization. APEL provides additional support for the Goal Question Metric (GQM [38]) model.

2000 PROMENADE [39] is the Process Oriented Modeling and Enactment of Software Development. It is a UML meta-model extension. PROMENADE supports active and pro-active controls (see JIL). Figure 1.31 shows a refinement hierarchy

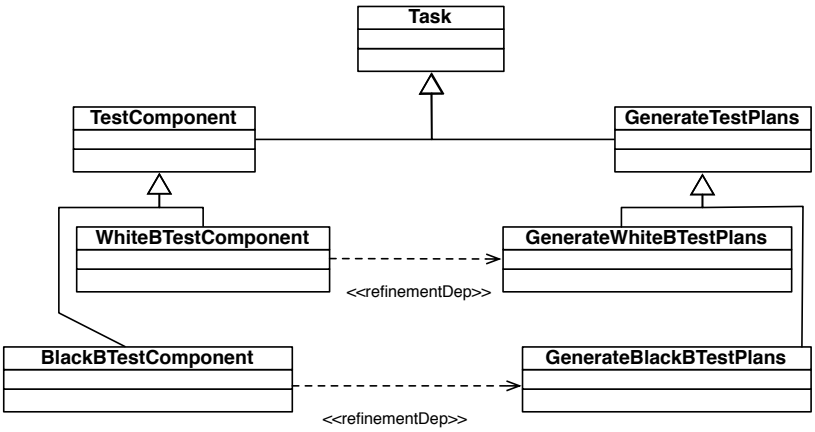


Figure 1.31: PROMENADE example (see [39], page 165)

for the task "TestComponent" and "GenerateTestPlans" in UML syntax.

2007 The Executable Process Modeling Language [40] is a formal, extensible Process Modeling Language based on some kind of Petri Nets semantics. Figure 1.32

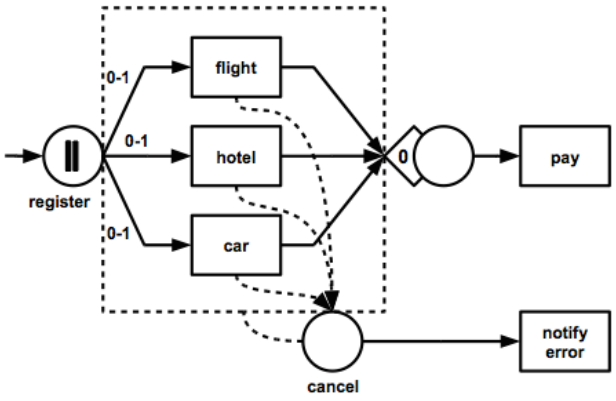


Figure 1.32: EPML example (see [40], page 13)

shows a travel reservation process.

2008 SPEM [6]: The Software & Systems Process Engineering Metamodel Specification is a MOF [41] instance developed by the Object Management Group (OMG). SPEM supports several diagram types. For details see 3.1.

This list of process modeling languages makes no claims of being complete but should give an impression of the diversity of process modeling languages that were defined. Several of these languages emerged and disappeared already. But none is widely accepted. A subset of these languages has formal defined semantics and is Turing complete. This stresses the argument that the expression power is not the critical issue to establish a process modeling language. If the language is Turing complete, has formal semantics (can be executed and analyzed) and has appropriate tool support the only reason not to use the language is a lack of sustainability and suitability (given a certain abstraction level) - in particular over time. This is in particular the case when only an abstraction of the language is relevant to describe an effect (e.g., an impact from one process element on another process element). This obviously includes the relevant meta-model of the language that defines syntax and semantics of the modeling language.

All these languages might be appropriate to describe some process elements but the chance is high that they are not appropriate over time and on a low abstraction level. Therefore these process modeling languages are not used directly in this approach. Instead a meta-modeling language is the key to include all these process modeling languages and all other potentially relevant languages on significant abstraction levels. Related meta-modeling approaches are introduced in the next section.

1.4.2 Meta-Modeling Languages

This section starts by introducing the meta-modeling approaches of the Generic Modeling Environment, Kermeta and Viatra. All approaches can handle a single meta-modelling layer. This means that the language to specify the semantics of the meta-model is fix. All approaches come with implementations to build and simulate the models.

The Generic Modeling Environment (GME 5) is a domain-specific, model-integrated program synthesis tool for creating and evolving domain-specific, multi-aspect models of large-scale engineering systems. Figure 1.33 shows screenshots of GME. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The techniques to build these models are (so called) hierarchy, multiple aspects, sets, references, and explicit constraints techniques [42]. The environment only focuses on building the models - or TestExecutedValueRofDifferencesMIL:more precise - their syntax/type/static aspect. Then so called model interpreters map the models to an implementation language to enrich them with some behavior. In later extensions a

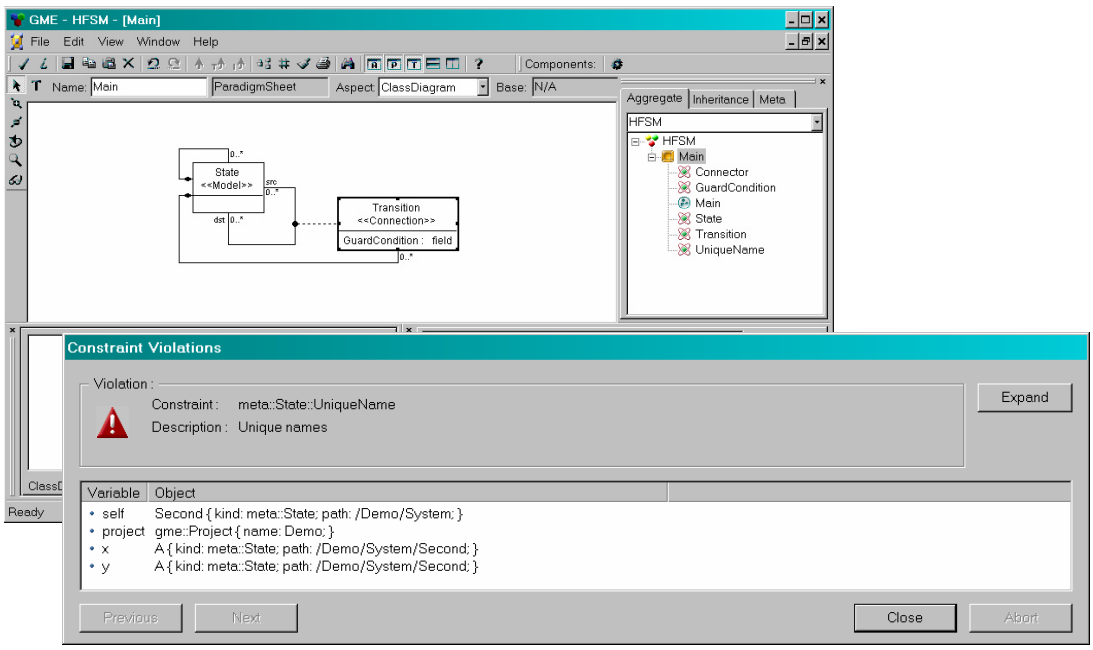


Figure 1.33: Generic modeling environment

graph transformation language is used to define semantics based on a language called "GReAT" (Graph Rewriting And Transformation [43]).

Kermeta [44] is a Mermaid-modeling approach dedicated to describe domain specific models emerging in a product design process in order to use model transformations to automate process activities/steps. Kermeta is an Essential MOF [41] compliant, can describe data types and comes with an own textual language to describe behavior. Listing 1.1 shows some example Kermeta code.

Listing 1.1: Example Code Kermeta

```
do // a loop for getting a text from an user
 var s : kermeta::standard::String
 from var found : kermeta::standard::Boolean init false
 until found
 loop
  s := stdio.read("Enter a text:\n-->")
  if s.size > 0 then found := true
  else stdio.writeln("ERROR - Empty text!")
  end
```

```
 end
 stdio.writeln("\n You entered: " + s)
end
```

The source code shows some object oriented language relationships. Another approach has been realized in the framework called Viatra [45], which is the abbreviation of "Visual Automated Transformations for Formal Verification and Validation of UML Models". In this framework textual graph transformations (based on GReAT) and abstract state machines are used to carry out graph transformations. Listing 1.2 shows some example code modifying a Petri Net. In this code a precondition (or Left-HandSite (LHS) pattern) and a post condition or (Right-HandSite (RHS) pattern) is presented concluded by an action sequence that is responsible for naming a place (in a Petri Net) in the same way as the control flow element was named. This name copying is defined in an imperative ASM rule called copyName, that is simply invoked here. It is worth pointing out that the LHS and RHS share the ControlFlow variable, therefore it will mean the same model element in both cases.

Listing 1.2: Example Code GReAT

```
gtrule transformCtrlFlw(out ControlFlow, in PetriNet)
= { precondition pattern unmappedCtrlFlw(ControlFlow)
= {'ControlFlow'(ControlFlow); neg find act_EdgeMap
   (ControlFlow, NoPetriPlace);}
 postcondition pattern mappedCtrlFlw
 (ControlFlow, PetriPlace, PetriNet) = {
  'ControlFlow'(ControlFlow);
  find activityEdgeMapping(ControlFlow, PetriPlace);
  find placeOfNet(PetriPlace, PetriNet);}
 action { call copyName(ControlFlow, PetriPlace); } }
```

The action sequence has access to the value of PetriPlace, which is the place created by the RHS (or more precisely, by applying the declarative part of the rule). The parameters of the graph transformation rule are the variables ControlFlow and PetriNet which are assigned in the pre- and post-conditions.

Besides these single layer meta-modeling approaches there exists a multiple layer meta-modeling approach defined by the OMG in form of the "Meta Object Facility". Figure 1.34 shows the Essential MOF (meta-)class definition, i.e., a class has a set of (ordered) attributes. This meta-model is the core meta-model for several OMG specifications like the Common Warehouse Meta-Model, the Unified Modeling Language, or the Software & Systems Engineering Meta-Model. The specification defines more or less in natural language the syntax of the language but gives a hint on the idea behind

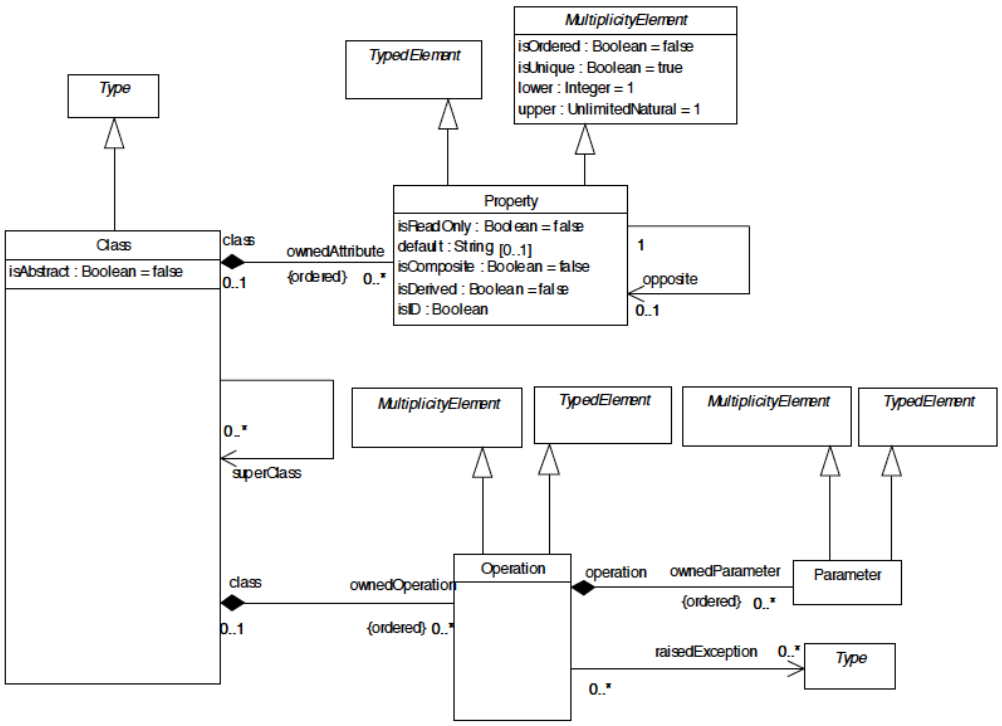


Figure 1.34: MOF

the approach how to define and relate multiple language layers.

The meta-modeling approaches define frameworks to define syntax and semantics for a single language including textual behavior for example in form of graph transformations. Additionally the MOF specification defines informally syntax and semantics of a multiple layer meta-modeling approach.

None of the previous approaches combines a rich, formal, and strongly typed but flexible meta-modeling language. Flexible in a way that the meta-modeling language can also be strongly typed on base of another meta-modeling language e.g., to synthesize meta-modeling elements. This is only possible with a multi-layer meta-modeling approach. Regarding the implementation of the approach it should base on standards whenever possible to ensure a maximal dissemination und support an arbitrary combination of textual and/or graphical representations (for all layers).

The next section discusses methods and techniques in the context of process optimzation complementing the approach.

1.4.3 Process Optimization

Synthesis[1] is the combination of elements to a new element. Model synthesis is required to build models or model parts automatically based on syntactical and behavioral requirements, e.g., resource consumption in relation to planned activities, tools and resources.

The counterpart of the synthesis is the analysis. Analysis is the systematical investigation of an element by dividing the element into its sub-elements without neglecting the sub-elements connecting/interlinking as a whole. Model verification is an analysis method to check (syntactical and) behavioral requirements of models and/or (meta-)models, e.g., possibility/probability to miss deadlines in relation to a set of planned activities, tools and resources. The process model optimization is often a combination of analysis and synthesis techniques.

There are four important questions when applying a method all focusing the single aspect of the method's suitability. First, the input prerequisites that must be fulfilled in order to apply the method. Prerequisites are, e.g., required model properties like relationships between variables must be linear, or normally distributed dependent variables. The required model properties often affect the model sustainability. Second, theoretical and practical limitations of the method, e.g., computation boundaries. Third, the effort that is required to interpret the results. Fourth, the precision of the results regarding the analysis question. The next sections introduce methods and techniques to synthesize, verify and optimize models shortly in relation to their suitability.

Synthesis

According to the definition above, model synthesis is the combination of model elements to construct a new model element. This can be for example a mathematical formula, tokens in a Petri Net, the synthesis of the net itself or the (meta-model) net semantics to explain certain aspects/effects in other models (often on base of surveyed data in the context of processes). The next section introduces several statistical methods which are widely applied in practice. The list of methods is not complete but covers the most applied methods and method categories.

Statistics Statistical methods and techniques often have strict prerequisites, e.g., regarding the variables' scales (nominal, ordinal, ratio), probability distributions or

[1] Defined by Hermann Kolbe to describe the combination of chemical substances to build a new substance.

dependencies (e.g., linear) between variables. So called "parameter free statistics" are used to keep these prerequisites as low as possible. Albeit the name might suggest that these methods have no prerequisites at all, they actually have prerequisites but not as strict as non free methods. Parameter free statistics don't restrict the statistical model structure but might be able to reveal some from the given data. Non parameter free statistics require a certain model structure, probability distribution, often a normal distribution.

There exist several parameter free statistics like median dividing the area of a distribution function into two equal parts, Kernel Density Estimation method to estimate the probability distribution of a variable and Cluster Analysis methods reveal potential clusters in a variable distribution. The next sections introduce exemplarily different statistical methods to compute relationships between variables. All examples in this sections are fictionalized simplifications for expository purposes and should not be taken as being realistic. The so called χ^2 independence test is introduced in the next section because it is the only test of this category to apply for any kind of relationship between two variables.

The χ^2 Independence Test The χ^2 test does not require any specific variable scale and is considered to be a parameter free test. A cross tabular is created in the first step of a χ^2 independence test. A cross tabular presents relationships between two nominal scaled variables. The total number of observations of a certain characteristic combination of the first variable (horizontally depicted in the cross tabulation) is mapped to the total number of observations of a certain characteristic of the second variable (vertically depicted in the cross tabulation) [46]. In the second step the expected values are computed assuming that the variables are independent and the values are equally distributed. The expected value is computed by multiplying the row sum with the column sum divided by the total number of observations. Figure 1.35 shows an example cross tabulation of the study [4] presenting the two variables "SUCCESS points" and "Number of employees". The cross tabulation of figure 1.35 shows that 41,1% of all projects have been conducted in companies with employees between 10 and 49. Assuming an independence of SUCCESS points and company size, 41.1% of the projects rated with 100 SUCCESS points should have been conducted in this company size class which equals 74 projects. The surveyed value is 89 projects. The χ^2 independence test utilizes these differences between expected and surveyed values. The smaller the difference the more likely is an independence, the larger the more unlikely. Whereby χ^2 is computed as follows:

$$\chi^2 = \sum_{i=1}^{I} \sum_{j=1}^{J} \frac{(n_{ij} - e_{ij})^2}{e_{ij}} \text{ with } i, j, I, J \in \mathbb{N}$$

			SUCCESS Points				
			100	90-99	68-89	<68	Total
Number of employees	> 249	Number	8	12	2	6	28
		Expected Value	14.28	7.85	2.78	3.09	28
		Percentage	28.57%	42.86%	7.14%	21.43%	100.00%
	50-249	Number	28	26	13	9	76
		Expected Value	38.75	21.31	7.54	8.40	76
		Percentage	36.84%	34.21%	17.11%	11.84%	100.00%
	10-49	Number	89	31	11	14	145
		Expected Value	73.94	40.67	14.38	16.02	145
		Percentage	61.38%	21.38%	7.59%	9.66%	100.00%
	< 10	Number	55	30	9	10	104
		Expected Value	53.03	29.17	10.31	11.49	104
		Percentage	52.88%	28.85%	8.65%	9.62%	100.00%
Total		Number	180	99	35	39	353
		Expected Value	180	99	35	39	353
		Percentage	50.99%	28.05%	9.92%	11.05%	100.00%

Figure 1.35: Cross tabulation SUCCESS points and number of employees

With i = cross tabulation column, j = cross tabulation row, e = expected value, n = surveyed value. The difference between surveyed and expected values is squared to avoid adding up of positive and negative values. The division by the expected values is done to normalize the values and to weight differences in relation to the total numbers. Hypothesis verification includes always the risk of errors. One distinguishes first kind errors (α-errors) and second kind errors (β-errors). If the so called null hypothesis (defined as contradiction to the working hypothesis) is rejected although the hypothesis is correct, it is called α-error. The significance level α describes the limit of first kind errors. Is a null hypothesis accepted although it is incorrect, it is called β-error. The interval of the χ^2 - distribution function has the value one. The α-error describes the relative size of the right part of the interval. If the computed χ^2 value matches the left part of this interval, the working hypothesis is rejected, otherwise it is accepted. This so called critical value is based on the so called freedom degree of the cross tabulation computed as follows: $f = (\text{number of rows} - 1) * (\text{number of columns} - 1)$. χ^2 is null, if the two variables are completely independent. If χ^2 rises, the dependence is more likely. Figure 1.36 shows some χ^2 distribution functions with independent freedom degrees and $\alpha = 5\%$, a critical value, and acceptation and rejection areas. The higher the freedom degree the more moves the mode of the distribution to the right. Only 20% of the expected values should be less or equal 5 and all expected values should be greater or equal 1 [47]. The main problem of the χ^2 independence test is already affected by this application recommendation. The classes of the cross tabulation are more or less (if the application recommendations are met) arbitrarily chosen - which is one important drawback. The second drawback is that the described relationship is weak, because the method simply states that the probability of a relationship is high

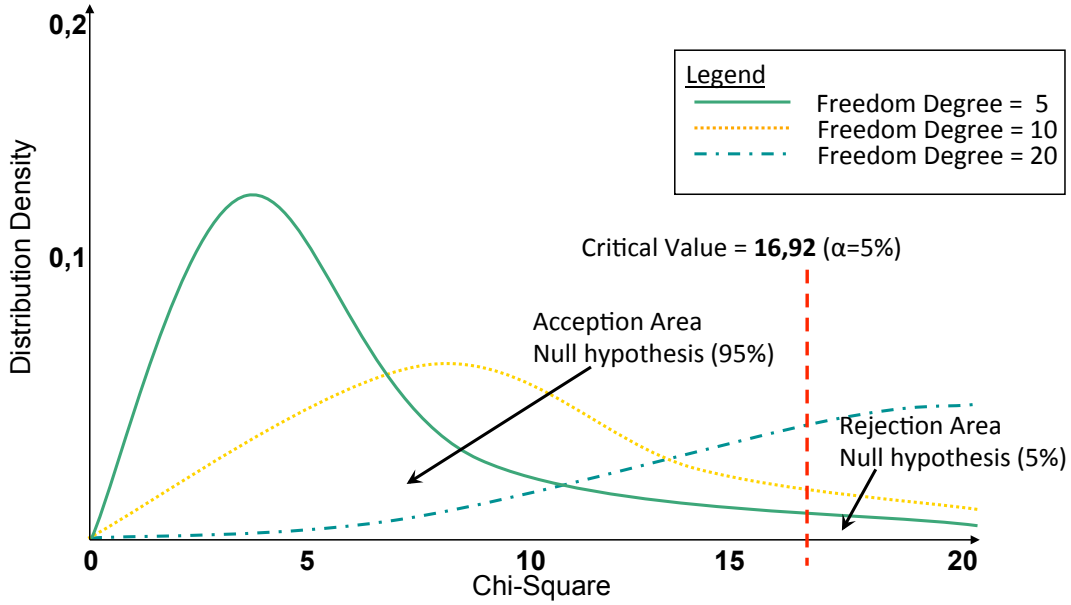


Figure 1.36: χ^2 - distribution function example

(or low). But it does not say anything else about the relationship. The next section introduces a method that computes the (linear) relationship strength of two variables.

Spearman Rho Spearman Rho is a parameter free association measurement for ordinal variables. The calculation of Spearman Rho is based on the so called rank correlation coefficient. Let there be two variables a, b with 9 observations that have been surveyed. Variable a is a software component complexity indicator. Variable b is the time required for testing activities for each software component. Table 1.1 shows the observations of a, b. The first step is done by computing their rank. Table 1.2

i	1	2	3	4	5	6	7	8	9
a_i	1	2,3	6,1	3	3	1	5	5	9,5
b_i	2	2,7	11,3	5	5,5	1,5	9,5	10	21

Table 1.1: Two example variables a,b and their observations

shows the rank computation of variable a. Table 1.3 shows the rank computation of Variable b. Several observations with the same value have a rank equal to the mean of the sum of their unnormalized ranks. The rank computation sorts the input values and assigns a rank according to the sort index. If multiple observations have the same

Index	Value	Rank	Normalized Rank
1	1	1	$(1+2)/2 = 1.5$
6	1	2	$(1+2)/2 = 1.5$
2	2,3	3	3
4	3	4	$(4+5)/2 = 4.5$
5	3	5	$(4+5)/2 = 4.5$
7	5	6	$(6+7)/2 = 6.5$
8	5	7	$(6+7)/2 = 6.5$
3	6,1	8	8
9	9,5	9	9

Table 1.2: Computing the rank of variable a

Index	Value	Rank	Normalized Rank
6	1,5	1	1
1	2	2	2
2	2,7	3	3
4	5	4	4
5	5,5	5	5
7	9,5	6	6
8	10	7	7
3	11,3	8	8
9	21	9	9

Table 1.3: Computing the rank of variable b

value, the rank is the division of the sum of the sorted indexes. The next step computes the differences between the ranks as presented in table 1.4. The third step computes the rank difference as follows:

$$\gamma_s \stackrel{df}{=} 1 - \frac{6\sum_i d_i^2}{n*(n^2-1)} \text{with } d_i = Rg(x^i) - Rg(y^i), \text{ with}$$
$$d_i = \text{rank difference}$$
$$n = \text{number of values}$$

Using the concrete observations we have $\gamma_s = 1 - \frac{6*1,5}{9*(9^2-1)} = 0.012346$. The values ranges from $[0, 1]$, whereby 0 indicates the strongest correlation and 1 the weakest. In the example case there is a strong relationship between the ranks of variable a and b, meaning that the ordering between observations of a and b is highly related.

Index	$d = Rg(a) - Rg(b)$	d^2
1	-0.5	0.25
2	0.5	0.25
3	0.0	0.00
4	-0.5	0.25
5	0.5	0.25
6	-0.5	0.25
7	0.5	0.25
8	0.0	0.00
9	0.0	0.00

Table 1.4: Computing differences between a and b

Regression Analysis The goal of the regression analysis [46] is the quantitative description of a relationship between a single independent variable and a set of dependent variables. Mathematically formulated: Let y be the dependent variable and $x_1, x_2, \ldots, x_n$ be a set of independent variables. The regression analysis tries to determine the function that describes:

$$y = f(x_1, x_2, \ldots, x_n) + e \text{ where}$$
$$e \text{ is the error (or residual)}$$

There are different forms of regressions that can be applied according to the parameters of the function f and prerequisites for $x_1, \ldots, x_n$ that must be fulfilled. $\underline{X}$ is the matrix consisting of the data values of $x_1, \ldots, x_n$. The linear regression analysis computes a linear function f to describe the relationship between the dependent variable and the set of independent variables. The surveyed data describes n equations of the following form:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \text{ with } i = 1, \ldots, n$$

The goal of the linear regression analysis is to determine β_0, β_1 and has the following assumptions:

- The random vector $\underline{\epsilon} = (\epsilon_1, \ldots, \epsilon_n)$ has an expected value of zero, i.e., $E(\underline{\epsilon}) = 0$.
- All ϵ_i are stochastically independent to avoid any relationship to Y.
- $\underline{X}$ is constant and has a rank of $n + 1$ which is required for a unique solution of the regression problem.

The computation of f is (normally) done by using the method of "least squares" that minimizes the squared sum of the residuals. The minimization of "least squares" is

i	a_i	b_i	b	a
1	1	2	1.88	0.12
2	2.3	2.7	2.91	-3.99
3	6.1	11.3	1.75	0.64
4	3	5	2.64	-2.92
5	3	5.5	2.13	-0.90
6	1	1.5	2.04	-0.54
7	5	9.5	1.87	0.16
8	5	10	2.36	-1.81
9	9.5	21	2.43	-2.08
Sum	35.9	68.5	2.25	-1.36

Table 1.5: Computation of a and b for example observations 1.1

computed as follows:

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - (a + bx_i))^2 \rightarrow min!$$

The set of given formulas determined by $\underline{X}$ can be transformed by partial differentiation into a set of normal equations. Then the searched so called regression coefficients a, b are:

$$b = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2}$$
$$a = \overline{y} - b\overline{x}$$

Let's take the example data of tabular 1.5 and determine a,b. Figure 1.37 shows a line plot (in blue) of the observations and the computed linear function $y = -1.36 + 2.25x$ (in black). The regression analysis is the first precise model that can be used for predictions. Normally one would compute a coefficient of determination (like R^2) showing how good the approximation is. There exist other regression variants like kernel regression (as a weighted form) up to spline functions. Nevertheless the prerequisites are high in all three statistical methods and they cannot be used on dynamic systems.

Factor Analysis The Factor Analysis tries to find a set of unobserved, uncorrelated variables in terms of linear combinations of a set of observed (presumably) correlated variables. Let $x_1, \ldots, x_n$ be a set of observable variables with means $\mu_1, \ldots, \mu_n$. Let's further assume we have factors $f_1, \ldots, f_m$ that describe the observations of $x_1, \ldots, x_n$

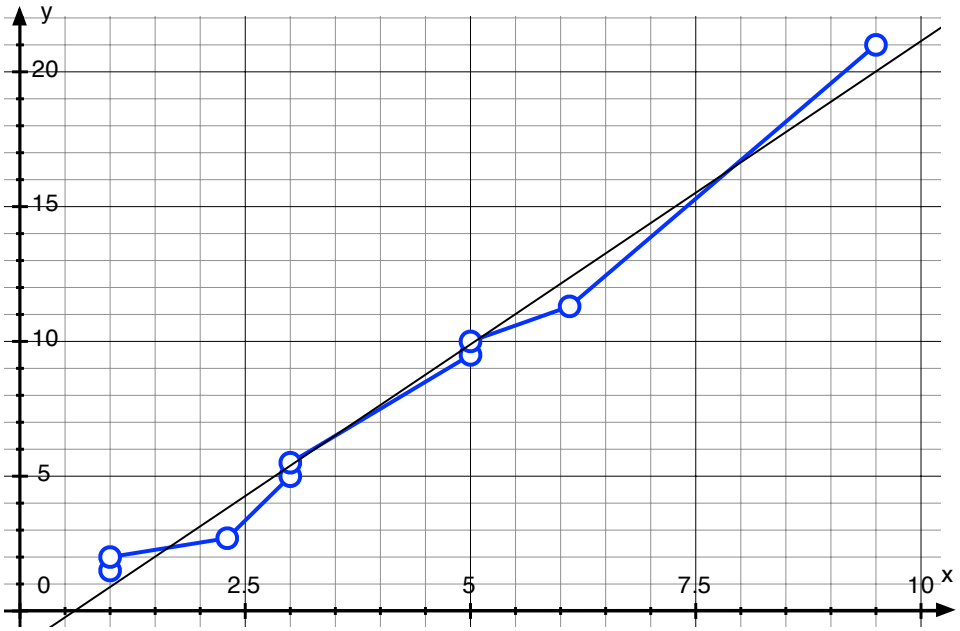


Figure 1.37: Linear regression analysis example

in the following way:

$$
\begin{aligned}
x_i - \mu_i = & l_{i,1} f_1 + \cdots + l_{i,k} f_k + \epsilon_i, \text{ with} \\
& i \in 1, \cdots, n \\
& k \in 1, \cdots, m \\
& l_{i,k} \text{ are some unknown constants} \\
& \epsilon_i \text{ are independently distributed error terms.} \\
& Variance(\epsilon_i) = \psi_i \\
& Co - variance(\epsilon) = \Psi \\
& \text{and } E(\epsilon) = 0
\end{aligned}
$$

In matrix terms it is written as $x - \mu = LF + \epsilon$. Any solution of the set of equations is defined as "factors" and $L = l_{1,1}, \ldots, l_{n,m}$ as so called "loading matrix". There exist different types of factoring approaches like the "Principal component analysis", which successively searches linear combinations of variables such that the maximum variance is extracted, the "Common factor analysis", which seeks the least number of factors, or the "Factor regression model", which is a combinatorial model of factor model and regression model. After the relationships between the variables have been computed they are often optimized. This is exemplarily shown in the next section.

1.4.4 Linear and Non-Linear Optimization

Linear and non-linear Optimization [48] tries to maximize (or minimize) a variable in a system of interrelated variables in relation to a given set of constraints.

Linear Programming and Nonlinear Programming

Linear programming requires that all relationships between the variables are linear. A linear program (LP) is a matrix $A \in \mathbb{R}^{m,n}$ with two vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. A solution of LP is a vector $x \in \mathbb{R}^n$ with non-negative entries that follows the linear constraints:

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + \ldots + a_{2n}x_n &\leq b_2 \\ \vdots \qquad\qquad \vdots \qquad\qquad &\vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n &\leq b_m \end{aligned}$$

The goal is to find the maximum of the scalar product $c^T x = c_1x_1 + \ldots + c_nx_n$. Assuming that the linear constraints are consistent there exist infinite possible solutions or at least one optimal solution.

A popular algorithm for numerical solutions of linear programming problems is the Simplex algorithm developed by George Bernhard Dantzig. The Simplex algorithm is done in two phases. Phase one tries to determine an initial solution or determines that the problem has no solution. One solution for this task is to introduce a variable z_i for every row i:

$$min\left\{ \sum_{i=1}^{m} z_i \middle| Ax + z = b, x, z_1, \ldots, z_m \geq 0 \right\}$$

The problem has the simple solution $(x, z) = (0, b)$ if $b \geq 0$. Furthermore is the target function restricted by $z \leq b$. Phase two tries to compute a better target function value by solving the linear equation system by applying the Gaussian elimination method iteratively. The following section sketches a small example:

Let's assume there are three employees, a developer, a tester, and a deployer e_1, e_2, e_3 who can be assigned in two projects p_1, p_2. All employees have a time frame to conduct their activities. The developer e_1 has 50 hours left, tester e_2 has 40 hours left, deployer e_3 has 60 hours left. To develop, test and deploy a function in project p_1 3, 2, and 2 hours are required. To develop, test and deploy a function in project p_2 4, 5, and 9 hours are required. A function of project A costs the client 400 Euros. A function

of project B costs the client 800 Euros. It is possible to construct the following linear constraints:

$$\begin{aligned} 3p_1 + 4p_2 &\leq 50 \text{ (developer)} \\ 2p_1 + 5p_2 &\leq 40 \text{ (tester)} \\ 2p_1 + 9p_2 &\leq 60 \text{ (deployer)} \\ p_1 \geq 0, p_2 &\geq 0 \end{aligned}$$

We would like to maximize the revenues z: max $z = 400p_1 + 800p_2$. This results in the Simplex Tableau 1.6: In the head are the non-basis variables (p_1, p_2) presented. The

	p_1	p_2	
-z	400	800	0
y_A	3	4	50
y_B	2	5	40
y_C	2	9	60

Table 1.6: Simplex tableau phase 1

basis variables (slack variables) y_A, y_B, y_C are presented in the first column. Phase 1 is done by setting the independent variables to zero (nothing is developed). In phase 2 a non-basis variable with positive target value is selected.

$$\overline{p_1} \leq min\left\{\frac{50}{3}, \frac{40}{2}, \frac{60}{2}\right\} = \frac{50}{3}, \quad \overline{p_2} \leq min\left\{\frac{50}{4}, \frac{40}{5}, \frac{60}{9}\right\} = \frac{60}{9}$$

The value of the target function would be $\overline{z} = 6666.67$ (p_1) or $\overline{z} = 5333.34$ (p_2). Therefore p_1 is selected. Now, y_A is exchanged with p_1, whereby $y_A = 50 - 3p_1 - 4p_2$ with $p_1 = 50/3 - y_A/3 - 4/3p_2$. Then x_1 is substituted in the other equations resulting in a new Simplex Tableau 1.7. In the second step the final non-basis variable with

	y_A	p_2	
-z	-400/3	800/3	-20000/3
p_1	3	4	50
y_B	-2/3	7/3	20/3
y_C	-2/3	31/3	80/3

Table 1.7: Simplex tableau phase 2

positive target value is selected.

$$\overline{p_2} \leq min\left\{\frac{50}{4}, \frac{20}{7}, \frac{80}{31}\right\} = \frac{80}{31}$$

This results in a target function value of max $z = 6666.67 + 2064.52 = 8731.17$. The simplex algorithm has polynomial-time average-case complexity under various probability distributions, even though it has exponential worst-case complexity. Khachiyan introduced 1979 an algorithm which solves LP in polynomial time (Khachiyans Theorem).

Nonlinear programming includes (as the name indicates) at least one non-linear function. Special variants are objective functions that are non-linear. If the objective function is concave, it a maximization problem, if it is convex it is a minimization problem. There exist several methods for solving non-linear problems, e.g., "branch and bound" tries to divide the non-linear problem space into linear approximations. Besides optimization it is often required to verify if a system (of variables and relationships) fulfills certain properties. This is exemplarily introduced in the next section.

1.4.5 Verification

Verification describes the task to determine if a given system has certain properties. Normally these properties are behavioral properties based on system snapshots. A system snapshot is a valuation of all variables for a given system state, e.g., after 5 steps. In order to determine a system state the model as well as the properties to be checked, must be formally defined.

Model Checking [1] relies on a formal system model. Initially this was a so called "Kripke-Structure", a minimal representation of the (reactive) system. Currently other representations are used (at least internally) that can be analyzed faster with a computer, but Kripke Structures are easier to understand and thus used for the example. Kripke Structures consist of a set of states and a set of transitions including a function that assigns each state a set of true statements. A calculation in the Kripke Structure is described by a transition path.

Definition Kripke-Structure: Let AP be a set of atomic assumptions. The Kripke-Structure M over AP is a tuple $M = (S, S_0, R, L)$ with:

(i) S is a finite set of states.

(ii) $S_0 \subseteq S$ is the set of initial states.

(iii) $R \subseteq S\chi S$ is a total transition relation, i.e., for all states $s \in S$ there is a successor state s' in S with $R(s, s')$.

(iv) $L : S \to 2^{AP}$ is a function that maps every state to a set of atomic assumptions.

A path π is represented by an infinite sequence of states $s_0 s_1 s_2 \ldots$ of the Kripke Structure with $\pi = s_0 s_1 s_2 \ldots$ and $s_i \in S$ connected by transitions.

To describe system states first order logic formulas are used and map every system variable $V = \{v_1, ..., v_n\}$ to a value of their domain. All mappings are elements of Ω. A system state s is represented by $s : V \rightarrow D$). Every Ω can be described by a formula, for example:

- $V = \{v_1, v_2, v_3\}$
- Ω: $\langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 \rangle$
- Some (true) formulas: $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$, $(v_1 = 2)$, $(v_1 > 0) \wedge (v_1 < 4)$

Figure 1.38 shows an example of a Kripke Structure. All formulas describe only true

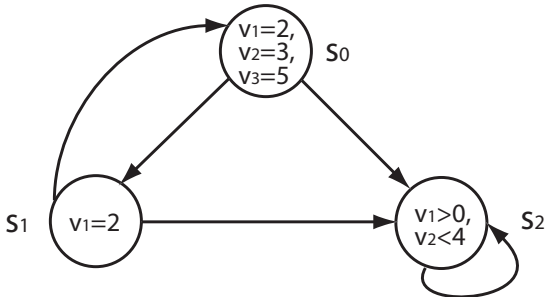


Figure 1.38: Kripke structure example

statements of Ω. We use the formulas to describe system states or sets of system states. Analog it is possible to describe transition sets. A transition switches between two states, the present state and the next state (V = present state and V' = next state). All variables of V, V' can be described with a formula, i.e., every variable v in V has a successor variable v' in V'. All transitions are captured by a transition relation $\mathcal{R}(V, V')$, e.g., $x' = x + 1$. These kind of first order formulas are now transferred into a Kripke Structure inductively on the structure. Exemplary the formulas S_0 and $\mathcal{R}$ are used as follows:

- $S_0(x, y) \equiv x = 1 \wedge y = 1$
- and $\mathcal{R}(x, y, x', y') \equiv x' = (x + y)\ mod\ 2 \wedge y' = y$.

Let $D = \{0, 1\}$. Then a Kripke Structure $M = (S, S_0, R, L)$ is derived with:

- The set of states S is equivalent to the set of Ω in V, which is an equivalent of the Cartesian product of the domain variables $D = \{0, 1\} \rightarrow S = D \times D = \{(0,0), (0,1), (1,0), (1,1)\}$.
- The set of initial states Ω of V that fulfill S_0 is $S_0(x, y) \equiv x = 1 \wedge y = 1 \rightarrow S = \{(1,1)\}$.
- Let s and s' be two states, then valuations of D are in the set $\mathcal{R}(s, s')$, if $\mathcal{R}$ is evaluated to true $\mathcal{R}(x, y, x', y') \equiv x' = (x + y)\ mod\ 2 \wedge y' = y$ is $R = \{((1,1),(0,1)), ((0,1),(1,1)), ((1,0),(1,0)), ((0,0),(0,0))\}$.
- "Labeling function": $L : S \rightarrow 2^{AP}$ is defined so that $L(s)$ consists of the atomic assumptions that evaluate s to true
  $R = \{((1,1),(0,1)), ((0,1),(1,1)), ((1,0),(1,0)), ((0,0),(0,0)) \rightarrow L((1,1)) = \{x = 1, y = 1\}, L((0,1)) = \{x = 0, y = 1\}, L((1,0)) = \{x = 1, y = 0\}, L((0,0)) = \{x = 0, y = 0\}$.

The only path (the only computation) in the Kripke Structure is
$(1,1)(0,1)(1,1)(0,1)\ldots$

Temporal logics are used to describe behavioral properties that should be verified. There exist two basic qualitative approaches of temporal logics according to their time representation. The first one is "Linear Temporal Logics" (LTL [49]) which has (as the name indicates) a linear understanding of time. The second one is "Computation Tree Logic" (CTL [50]) which is in contrast to LTL able to formulate statements on branching time. The third one is CTL* which is a super-set of LTL and CTL combining path quantifiers of LTL and temporal operators of CTL. These three temporal logics have a qualitative notion of time. Quantitative statements, e.g., "in 2 seconds", are not possible. Quantitative statements are possible with special logics like Timed CTL and Duration Calculus. There exist also more user-friendly temporal logics like symbolic timing diagrams or life sequence charts. Nevertheless for a basic understanding of model checking LTL is sufficient. LTL combines classic logical statements with the following temporal operators:

Next: $\bigcirc(a > 1)$: a has a value > 1 in the *next* step. The LTL formula is true for a given path $(s_0, s_1, \ldots)$ if and only if $a > 1$ is true for s_1.

Always: $\Box(a)$: a is *always* true. The LTL formula is true for a given path $(s_0, s_1, \ldots)$ if and only if a is true for all $(s_0, s_1, \ldots)$.

Eventually: $\Diamond(a)$: a is eventually true (in the following (possible infinite) states). The LTL formula is true for a given path $(s_0, s_1, \ldots)$ if and only if a is true at least for one s_i with $i \in \mathbb{N}_0$.

Until: $(a)\mathcal{U}(b)$: a is true until b is true. The LTL formula is true for a given path $(s_0, s_1, \dots)$ if and only if there exists a $i \in \mathbb{N}_0$ that b is true in $(s_i, s_{i+1}, \dots)$ and for all $j \in \{0, \dots, i-1\}$ a is true for the path $(s_0, s_1, \dots, s_j)$.

The semantic of LTL operators is presented in figure 1.39. Model Checking tries to

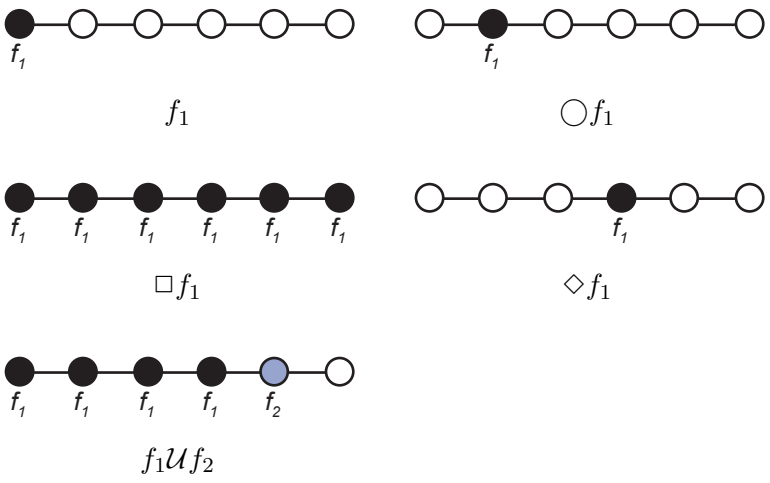


Figure 1.39: LTL formulas and their semantics in the calculation tree

identify the set of initial states $s \in S_0$ that are true for a given temporal formula f or differently formulated $\{s \in S_0 | M, s \models f\}$y marking the sets in the Kripke Structure. In the last years Kripke Structures have often been substituted by Ordered Binary Decision Diagrams that can often be computed faster by a computer. The Model Checking process is simple in general. Unfortunately one will often face the so called "State Explosion Problem", because the computation tree or similar representations are too large to be computed. The current research tries to reduce the computation effort by using partial order, abstractions or symmetries of the model [51].

Satisfiability solvers try to determine, if the variables of a given (temporal) Boolean formula can be assigned in such a way that the formula can be evaluated to true. This is normally done by a backtracking algorithm which chooses a literal, assigns a truth value, then simplifies the formula and checks the satisfiability. If the simplified formula is satisfiable, the original formula is also satisfiable. If the simplified formula is not satisfiable the opposite truth value is assumed.

Theorem Proving in general tries to prove mathematical theorems by applying rules of a given calculus for example first order logics. (Semi-)Automatic approaches are

supported by a computer by using term rewriting engines and tableaux provers as well as various decision procedures. The methods that were previously introduced were all "complete". They had several prerequisites to be applied (e.g., a normal distribution) and also their results are often restricted (e.g., they can only describe linear relationships). Additionally some had severe computation boundaries. The next section introduces an approximative and "incomplete" method (more or less) without these restrictions.

1.4.6 Stochastic Optimization and Metaheuristics

The analysis should cover static as well as dynamic model aspects. Traditionally, statistics are used to describe relationship between variables, formal verification methods are used to check model properties (over the time) and linear and non-linear programming methods are used to compute optimizations. All these methods have in common, that they can not be applied in general. The reasons are manifold, like normal distribution of the variables as prerequisite, linear relationships between independent and dependant variables without a temporal dimension as result constraint, and only non-concurrent computations.

Van der Aals et. al. have used different algorithms to mine process descriptions from event logs in form of business process models, like α algorithms [52]. Development process models require more flexibility. The first and most important analysis task is the synthesis. The synthesis should in particular cover the generation of complete models. The models should be build as explicit extraction of experiences/surveyed data. This targets the area of genetic programming which was left out on intention previously, because it will be covered in detail in section 1.4.3. Genetic programming focuses inherently on emphasizing the most significant data. It is possible to guide the search very flexible and a "warm start" of the algorithms is possible (i.e., intermediate results can be stored and used in new computations). The computation can be done distributed on a set of computers including a flexible memory footprint. Last not least the other approaches can be easily embedded into genetic programming.

The optimization approach in this thesis focuses on a general (meta-)model's heuristic exploration with genetic algorithms and genetic programming techniques. It's the first step (towards more specialized algorithms) to explain the unexplained, has no restrictions, is flexible and robust. The focus lies on a flexible process model optimization to explore process models. The other interesting aspect is a kind of symbiotic relationship between metaheuristics and RMOF, whereby RMOF guides effectively the learning process of the metaheuristics by e.g., model fitting (=what parts of the models where significant with respect to RMOF semantics). The same flexibility is required for the models itself to capture all activities and artifacts of development processes in dif-

ferent abstractions and viewpoints. The core of the models is RMOF as meta-language nucleus, which is defined in the next chapter.

2 Rich Meta Object Facility

> "The most evident difference springs from the important part which is played in man by a relatively strong power of imagination and by the capacity to think, aided as it is by language and other symbolically devices."
>
> Albert Einstein

A process model is an abstraction of a development process. The language used to describe the process model must be precise enough to address all development questions to achieve the defined process goals. The language needs to be as minimal in their syntax and as abstract in their semantics as possible that it covers only the syntax and semantics required to understand the implications on the process goals and to use it efficiently as planing instrument to guide/optimize the development. Available process modeling languages often lack the required flexibility, suitability and sustainability. This is the case because their syntax and semantics is inherently subjective and will likely even change over time (e.g., due to new members, tools and methods, or even development languages). This applies in particular for their adequate abstraction level during the development (a design artifact looks different in different development steps and different from different perspectives e.g., a designer perspective or a tester perspective). Last not least the process goals often change over the time, e.g., the statement coverage in the testing phase should be increased. This raises the question up to which level it is required to "understand" (=model and analyse) all test inputs and interactions (e.g., requirements, models to be tested and their generated source code, test tool interactions) including their syntactical and semantical dimensions. Therefore instead of a single process modeling language meta-process modeling languages were discussed in section 1.4.2.

This chapter defines a formal enrichment of the Meta Object Facility meta-modeling language/system to instantiate formal MOF based specifications. The meta-modeling approach is summarized in figure 2.1. The first language layer (L_0) consists of the meta-modeling approach of the Object Management Group complemented by UML2 dynamics (presented in green, describing the syntactical part of RMOF) and is com-

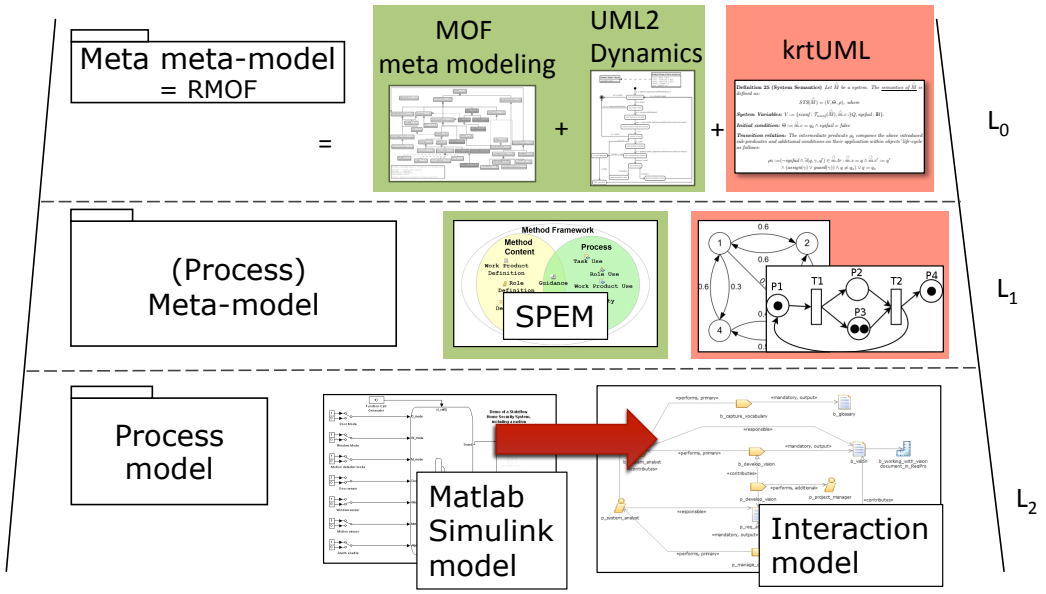


Figure 2.1: RMOF meta-modeling

plemented by krtUML (presented in red, describing the formal semantics) formalizing both. The MOF relevant data structures are described in section 2.1 with references to the origin OMG specifications. These data structures are complemented by expressions 2.2 and algorithms 2.3. The second language layer (L_1) describes the process modeling language as instance of the first language layer. The Software Process Engineering Meta-Model (SPEM see section 3.1) is partially instantiated and extended in this section to described tool interaction models and Matlab Simulink Models. This section concludes with a set of requirements of meta languages that no introduced meta language is able to fulfill and will introduce a meta-language that fulfills the requirements in the next sections.

The OMG provides an informal, incomplete language framework with the Meta Object Facility (MOF [41]). By formalizing and complementing MOF, a formal integration platform is available to formalize MOF based specifications successively based on step-by-step consolidated language elements and to define, synthesize and analyze process modeling languages as required.

Specifications of the Object Management Group (OMG) are often widely applied e.g., the Common Warehouse Meta-model(CWM [53]), the Software Process Engineering Meta-model(SPEM [6]), and the Unified Modeling Language(UML [54, 55]). All of these languages are incomplete and informal leading to interpretation gaps. Formal pa-

pers bridge these interpretation gaps like [56, 57, 58]. The papers often concentrate on a subset of the original specification and use different notations to formalize the subset. This makes it often difficult to combine papers to form a complete and consistent yet formal language which would be necessary to apply the specifications appropriately. Therefore a language is required to combine different formal methodologies. This is not a trivial task, because there exist often good reasons to use a certain formalism, like intuitive descriptions, e.g., Petri Nets describing Activity Charts. The identified integration language must be powerful enough to describe required syntax and semantic (e.g., in UML it is necessary to express constraints, compute derived values, and operations). The language should be minimal to support interpretation and simplify mappings to other formal languages or implementations (e.g., a model checker input language).

In contrast to map informal specification elements onto a formal language, Damm and Pnueli et. al. define in [59] discrete-time UML semantics for concurrency and communication in safety-critical applications, (e.g., typing, events, triggered operation calls). The semantics of all UML languages can be mapped to this language. This chapter introduces an extension of MOF called Rich Essential Meta Object Facility (RMOF). Compared to other approaches like [60, 61, 62], relevant parts of the MOF and UML2 specifications are modified and extended in a conservative way supporting the knowledge transfer between researchers and users. The (core) formal semantics of RMOF is an extension and adaptation of *krtUML* complemented with a flexible mapping framework embedded into MOF to define syntax, semantics of different languages and specify relationships among them.

The chapter starts in section 2.1 with an introduction of the data structures covering simple and complex types as well as mechanisms to structure and instantiate them. The second section 2.2 introduces operations covering, e.g., arithmetic, collection type management, and layer management. The third section 2.3 introduces the definition of algorithms, e.g., states, transitions, and effects using the defined expressions of 2.2. Elements of MOF [41], the UML Infrastructure specification [54] and the UML Superstructure specification [55] are introduced by their specification number. Section 2.4 introduces so called "observer" to define and check temporal properties and section 2.5 concludes this chapter with an introduction of the implemented RMOF environment.

2.1 Data Structures

This section introduces the data structures of the RMOF core meta model $\mathring{m}$ as subset of MOF [41] (compliant to "Essential MOF"), the kernel package, the behavior package and state machine package of the UML2 Super Structure [55]. The UML2 and MOF

classes are prefixed in the RMOF specification by their UML2/MOF ID. MOF and UML2 are formalized using *krtUML* approach and complemented mainly regarding behavior. Figure 2.2 shows an overview on the data structures. Green classes refer to data structures relevant class parts, red classes refer to algorithm relevant class parts. Each class is separated into three segments. The class name is written in the first segment. If the class is abstract, the class name is written italic. Inheritance between classes is depicted by an arrow with an unfilled head, whereby the inheriting class points to the class it inherits from. The second segment contains the class attributes. The prefix "$< s >$" stands for "shared", "/" stands for derived. If an attribute has a complex type (referring to a set of other objects) and is not shared, it is "owned" by the object. Owned objects are destroyed when the owning object is destroyed. Shared objects are not destroyed. A derived object value is computed by a State Machine. After the optional prefix the attribute name is specified, with an optional multiplicity in squared brackets (the default is "[1..1]"), a optional comment in round brackets separated with ":" from the attribute type. The attribute name can be a simple type or a class name (of the same layer). The attribute type can be suffixed by an initial default value which is an element of the specified type. The third segment specifies class operations with their State Machines. Each operation can have multiple arguments with a direction, i.e., in, out, inout, and return.

Motivation to minimize the data structures is to reduce the effort that is required to map RMOF to a platform (like Java or Objective C). The data structures must be powerful enough to express everything that is necessary to describe the intended semantics. All classes are introduced by their identification number corresponding to the ID of the UML superstructure specification [55]. The ID is followed by the instantiated class name and set of attribute valuation pairs (attribute name, attribute value). The specification is complemented by a short description.

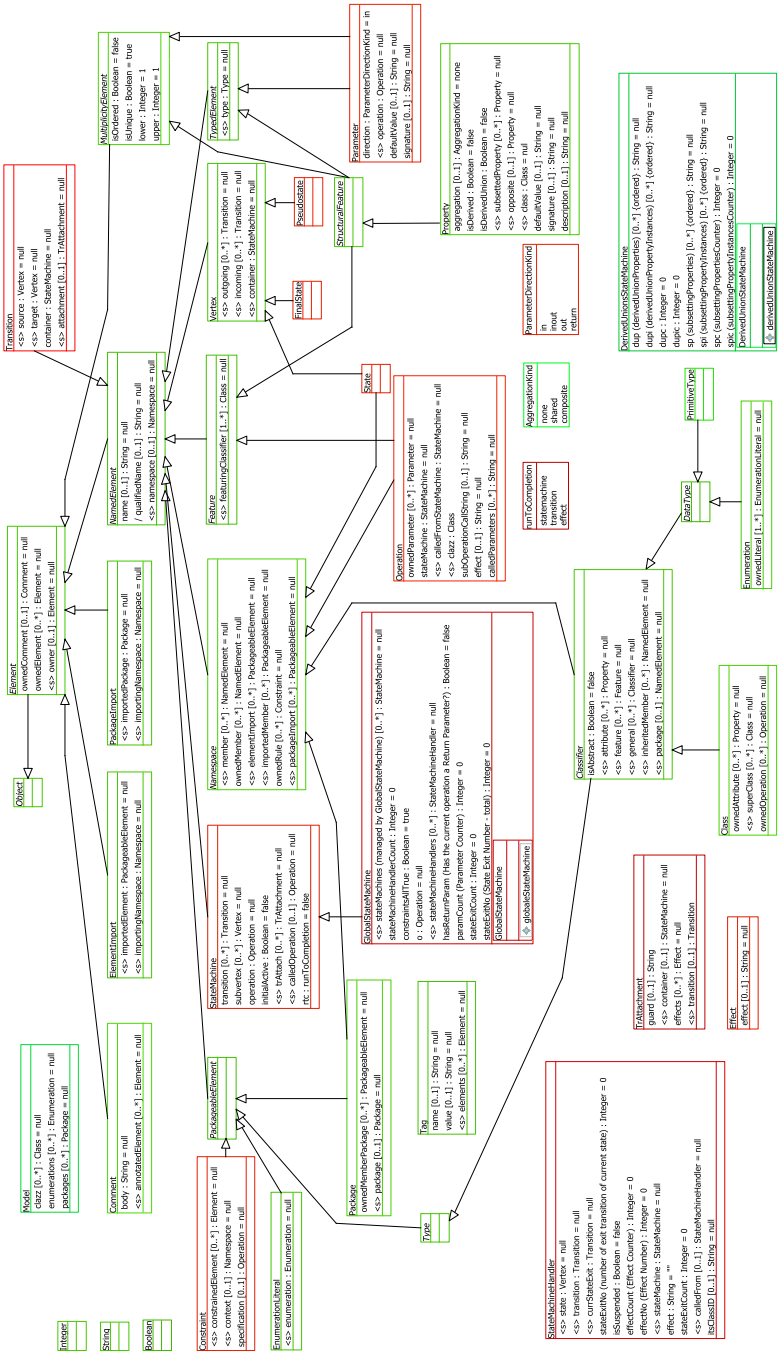


Figure 2.2: Data structures (green = static, red = dynamic)

Besides simple types RMOF introduces the following collection types:

Collection Type	**Written as**	**isOrdered**	**isUnique**
set	{...}	false	true
ordered set	[...]	true	true
bag	{{...}}	false	false
sequence	[[...]]	true	false

All constructs are specified as meta model instances to show their reflectiveness based on the following EBNF syntax:

```
DigitNotZero     ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
Digit            ::= '0' | DigitNotZero;
NatNum           ::= Digit | (DigitNonZero {Digit});
Letter           ::= 'a' | ... | 'Z';
String           ::= Letter {Letter};
Reference        ::= NatNum {'.' NatNum} ('-' NatNum)
                     | ('[P' NatNum ']');
Identifier       ::= String {'::' String};
Value            ::= ('"' String '"') | NatNum;
Values           ::= ('[' Value {',' Value} ']')
                     | ('[[' Value {',' Value} ']]')
                     | ('(' Value {',' Value} ')')
                     | ('((' Value {',' Value} '))')
                     | ('{' Value {',' Value} '}');
Attribute        ::= '(' String ',' Values ')';
Attributes       ::= Attribute {',' Attribute};
InstanceValues   ::= '=' (Identifier, {Attributes})';
Description      ::= String {'(' | ')' | '_' | ',_' | String
                     | '.'};
Instance         ::= Reference Identifier InstanceValues
                     Description;
Model            ::= Instance { Instance };
```

A specification access is realized by using the introduced non-terminals of the EBNF, e.g., "Description" provides a set of all descriptions. In the following text the data structures of RMOF are specified with respect to this EBNF.

RMOF is a framework to define multiple languages. Each language is contained in an RMOF layer. A layer consists of a name, a meta layer reference, a set of instances, a set

of primitive types, a mapping of instances to primitive types, and a set of operations. Layers are connected via meta layer references. Each instance (of the layer instances) refers to the meta-instance of the meta-layer and a set of attribute, value pairs, formally defined:

Definition 1 (Layer) *Let $N = \{a, \ldots, Z, 0, \ldots, 9, ::\}^*$ be a set of names/IDs. A layer l is a tuple ($l.n$, $\widehat{l}.n$, $l.I$, $l.T$, $l.A$, $l.O$), with*

- $l.n \in N$ *: A layer name.*
- $\widehat{l}.n \in N$ *: A reference to the (meta-)layer.*
- $l.I$*: A set of instances. Each* $(i.n, i.\widehat{n}, i.A) \in l.I$ *consists of*
  - $i.n \in N$ *: An instance name, whereby all names of a layer are pairwise disjoint,* $\forall (i_1.n, i_1.\widehat{n}, i_1.A), (i_2.n, i_2.\widehat{n}, i_2.A) \in l.I : i_1.\widehat{n} = i_2.\widehat{n} \wedge i_1.A = i_2.A \Rightarrow i_1.n = i_2.n$.
  - $\widehat{i}.n \in N$ *: A (meta-)instance reference.*
  - $i.A$ *: A finite, non-empty set of attribute, value pairs. Each* $(a.n, a.v) \in i.A$ *is a pair, with* $a.n \in N$*, and* $a.v$ *is build upon N in the way that* $a.v$ *is an element of N indirectly typed by the meta layer platform types or* $a.v$ *is a so called multivalued attribute formed on N as set {}, ordered set {{}}, bag[], or sequence[[]], e.g., 'a', '{a,b1::a}, '[{a,b1::a},{a,b1::a},{b2,c}]'.*
- $l.T$*: A set of platform types (e.g., Boolean, Integer, Float) with appropriate domain definitions (e.g.,* $\mathbb{B}, \mathbb{Z}, \mathbb{R}$*).*
- $l.A \subseteq \{i.n | (i.n, \widehat{i}.n, i.A) \in l.I\} \rightarrow l.T$ *: A set of type assignments.*
- $l.O$*: A set of platform operations (see Def. 14 - Def. 22).*

■

Layers and layer elements are connected via instances. Layers are described as collection of instances of a meta-layer. All instances base on type instantiations (of so called "platform types", i.e., types like Boolean with type domain definitions like {*false*, *true*}). All attributes can have values that are instances of their meta-instances instantiating types and thus elements of their value domains.

Definition 2 (Instance) *Let* $l_1 = (l_1.n, \widehat{l_1}.n, l_1.I, l_1.T, l_1.A, l_1.O)$, $l_2 = (l_2.n, \widehat{l_2}.n, l_2.I, l_2.T, l_2.A, l_2.O)$ *be two layers. Layer* l_1 *is instance of* l_2 *(written* $l_1 \dashrightarrow l_2$*), iff*

- *The meta-layer of* l_1 *is referencing* l_2*, i.e.,* $\widehat{l_1}.n = l_2.n$.

- *All instances are instantiated as follows:*
  $\forall(i_1.n, \widehat{i_1}.n, i_1.A) \in l_1.I\ \exists(i.n_2, \widehat{i_2}.n, i_2.A) \in l_2.I$*:*
  - *The meta-instance of* i_1 *is referencing* i_2*, i.e.,* $\widehat{i_1}.n = i_2.n$*.*
  - *All attribute value pairs of* i_1 *are instances of type introducing elements of* i_2*, i.e.,* $\forall(a_1.n, a_1.v) \in i_1.A\ \exists(a_2.n, a_2.v) : a_1.n = a_2.n \wedge \ldots$
    - *(i)* $\wedge\ (a_2.v, t) \in l_1.A \Rightarrow a_1.v \in t$*, instance of a platform type.*
    - *(ii)* $\wedge\ (a_2.v, ct)$ *and* ct *is a collection type* $\{\} \ldots [[]] \Rightarrow a_1.v$ *is element of this collection type domain (see Def. 18 - Def. 21).*
    - *(iii)* $\wedge\ a_2.v \in \widehat{i} \wedge \widehat{i} = \{i_2.n | (i_2.n, \widehat{i_2}.n, i_2.A) \in l_2.I\} \Rightarrow a_1.v \in \{i_{1,1}.n | (i_1.n, \widehat{i_1}.n, i_1.A) \in l_1.I \wedge (i_{1,1}.n, a_2.v) \in i_1.A\}$*, instance of a class type, if the transitive closure of* i_2 *has at least one element that is an instance of a platform type (1) or collection type (2).*

■

Each layer is a snapshot of its meta layer. A core layer is a special variant of a layer reflexively instantiating itself.

Definition 3 (Core Layer) *Let* l *be a layer.* l *is called a core layer iff* $\exists l' \in L : l' \dashrightarrow l \wedge l' = l$*, meaning all instances of* l *are described as instances of* l *itself (written* $\overset{\circlearrowleft}{l}$*).* ■

Each layer can be identified as Meta Model, Model, or Object Model according to the number of supported instantiations.

Definition 4 (Meta Model, Model, and Object Model) *Let* L *be a set of layers. Layer* $l \in L$ *is called*

- *Meta Model Layer, if the layer can be instantiated at least twice with different layers (if* $\exists l_2, l_3 \in L : l_3 \dashrightarrow l_2 \dashrightarrow l \wedge l \neq l_2 \wedge l_2 \neq l_3 \wedge l \neq l_3$*).*
- *Model Layer, if the layer can be instantiated once with a different layer and* l *is not a core layer (if* $\exists l_2 \in L : l_2 \dashrightarrow l \wedge l \neq l_2$*).*
- *Object Layer, if the layer cannot be further instantiated and* l *is not a core layer (if* $\nexists l_2 \in L : l_2 \dashrightarrow l$*).*

■

A set of layers forms a so called "System". A System consists of a core layer, (possibly) multiple meta-layers and one object layer.

Definition 5 (System) *Let M be a finite set of layers. M is called a System, if all layers in M are linked, meaning $\forall l \in M\ \exists \widehat{l} \in M : l \dashrightarrow \widehat{l}$ and M consists of one core layer, arbitrary meta models, a model and an object model (M is written $\mathring{m}$). M is well-formed, iff there exists maximal one instance layer of each layer i.e. $\forall l, l_2, l_3 \in M : l_2 \dashrightarrow l \wedge l_3 \dashrightarrow l \Rightarrow l_2 = l_3$.* ■

Each layer can introduce (platform-)types and all layer elements are strictly typed with respect to its meta layers.

Definition 6 (Layer Instance Typing) *Each layer $l = (l.n, \widehat{l}.n, l.I, l.T, l.A, l.O)$ defines the following types*

$$T(l) := l.T \cup \{t_{i.n,c} | (i.n, \widehat{i}.n, i.A) \in l.I \ \wedge c \in \{p, s, oS, b, q, S, OS, B, Q\}\}$$

■

Index p indicates a platform type value (e.g., Boolean with domain $\mathbb{B}$). Index of collection types (and their domains) are written as follows:

- 's' for set: $\{c, a, b\}$
- 'oS' for ordered set: $\{\{a, b, c\}\}$ including some kind of ordering relation R that is reflexive, transitive, and anti-symmetric.
- 'b' for bag: $[a, a, b, d, d, \ldots]$ as folded multi-set variant.
- 'q' for sequence: [[a, a, b, b, b,]] as ordered bag.

Every collection type can be instantiated by building (a set of) a set, ordered set, bag, or sequence of the corresponding primitive type. The domain of the type equals, e.g., $2^{t_{c,S}}$, if c is the non-collection type and the collection type is a set. Capital letters {S, OS, B, Q} are permutation sets of the collection types.

Definition 7 (Primitive Operation Typing) *For each $o \in l.O$, $type_{par}(o) = T_1 \times \ldots \times T_n$ denotes the parameter type where $T_i \in T(l)$ is the type of the i-th parameter and $type_\tau(o)$ denotes the value of f. The type of o is $type(o) = type_{par}(o) \mapsto type(o)_\tau$.* ■

Definition 8 (Expression Typing) *Expressions are inductively defined as follows:*

- *Navigation expression: $expr ::= \tau.a$, where $\tau \in \bigcup_{a \in i.A, (i.n, i.\widehat{n}, i.A) \in l.I}$ with $type(\tau) = T_{c0}$ and $a \in c_0.attr$, with $type(expr) := type(a)$.*

- *Primitive operation application:* $expr ::= o(expr_1, \ldots, expr_n)$*, where* $expr_1, \ldots, expr_n$ *are expressions,* $o \in l.O$*, and* $type(expr_i)$ *matches the type of the i-th parameter of* $o, 0 < i \leq n$*, with* $type(expr) := type_\tau(f)$*.*

■

Definition 9 (Guard Typing) *A guard* $[expr]$ *type is defined as* $type(expr) = \mathbb{B}$*.* ■

Definition 10 (Instance Typing) *'type' is used to denote the type of instances defined as follows:*

- *the type of all instances is the meta-instance* $\forall i = (i.n, \widehat{i}.n, i.A) \in l.I : type(i) := \widehat{i}.n$
- *the type of all instance attributes is a value of type* $\forall i_a = (a.n, a.v) \in i.A : type(i._a) :=$
$$\begin{cases} t & \text{if } \{(a.n, t), (upperValue, 1)\} \subseteq \widehat{i}.n \\ t_s & \text{if } \{(upperValue, *), (isOrdered, false), (isUnique, true)\} \subseteq \widehat{i}.n \\ t_b & \text{if } \{(upperValue, *), (isOrdered, false), (isUnique, false)\} \subseteq \widehat{i}.n \\ t_{oS} & \text{if } \{(upperValue, *), (isOrdered, true), (isUnique, true)\} \subseteq \widehat{i}.n \\ t_q & \text{if } \{(upperValue, *), (isOrdered, true), (isUnique, false)\} \subseteq \widehat{i}.n \end{cases}$$

■

For each type $t \in T(l)$, there exists a designated element $\mathsf{nil} \in t$ as default value and $\perp$ as error value. Based on the definitions 1-10 the core meta model $\overset{\circlearrowleft}{m}$ is introduced. The data structures of $\overset{\circlearrowleft}{m}$ mainly introduce types, structuring, and instantiation mechanisms. The chapter starts with simple types in section 2.1.1, followed by complex types in section 2.1.2 and instance specifications in section 2.1.3. In order to manage the introduced data types the definitions are complemented by additional informations in section 2.1.4 like comments (and especially naming mechanisms in section 2.1.6), structuring informations in section 2.1.5, and naming in section 2.1.6. The section is complementd by the definition of the core meta model and the introduction of configurations (variable snapshots) in section 2.1.7.

2.1.1 Simple Types

Elements that can only be instantiated by a single value are considered simple types. This includes for example primitive types like Boolean and user defined types in form of enumerations. Figure 2.3 presents all simple types including their generalization hierarchy. Aggregations and relations are additionally presented besides their attributes.

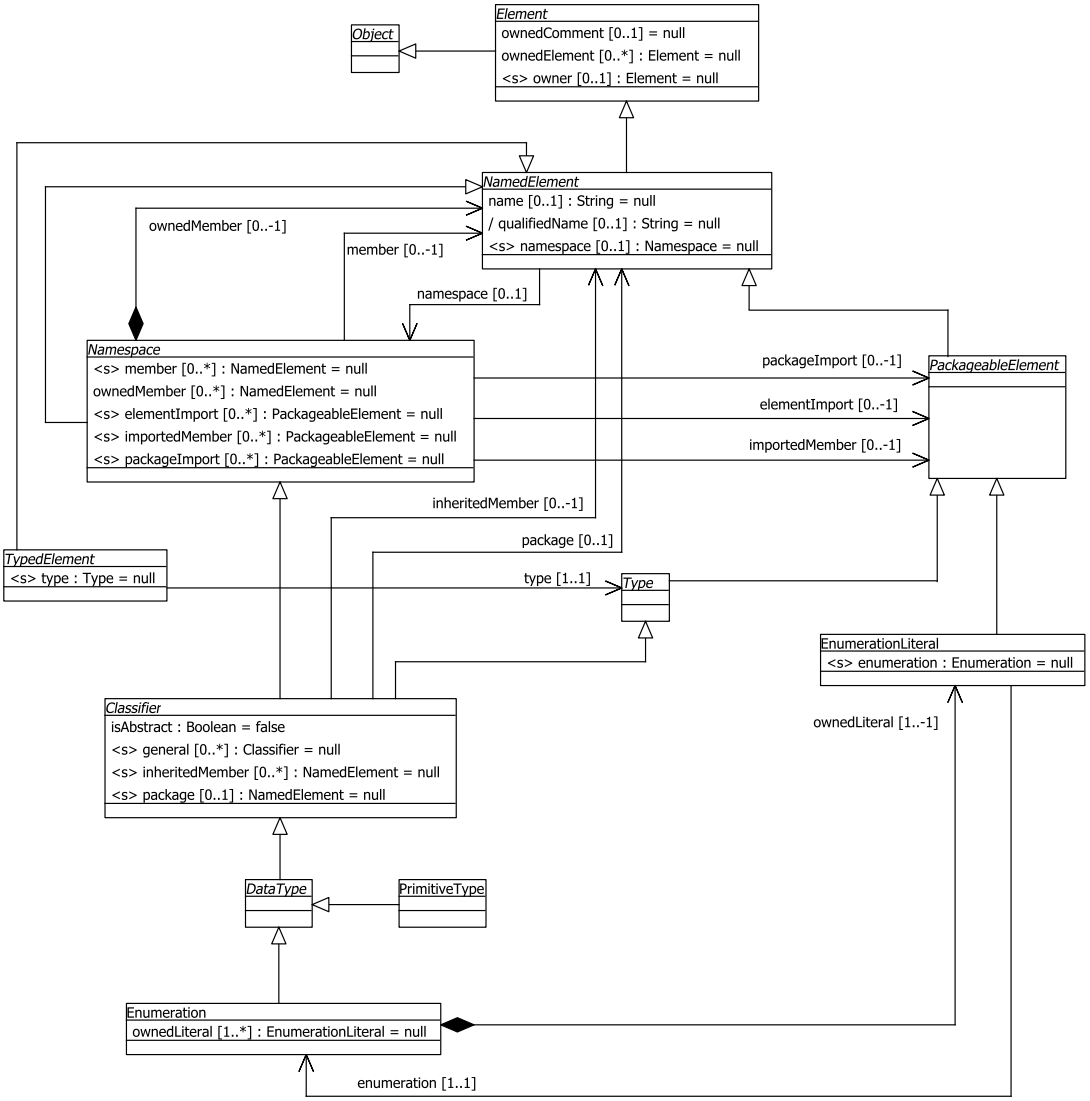


Figure 2.3: Class diagram of simple types

Enumerations are user-defined data types of a set of valid so called "enumeration literals". A variable of Enumeration X has a value of $x_1, \ldots, x_n$ if $x_1, \ldots, x_n$ are EnumerationLiterals of X.

7.3.16 Enumeration = (Class,{(ownedAttribute, (ownedLiteral)), (generalization, {Datatype})}) is a data type whose values are enumerated in the model as enumeration literals. Enumeration is a kind of data type, whose instances may be any

of a number of user-defined enumeration literals.

- ownedLiteral = (Property,{ (aggregation, composite), (subsettedProperty, {Element::ownedMember}), (type, EnumerationLiteral), (upperValue, *)}) specifies the ordered set of literals for this enumeration.

7.3.17 EnumerationLiteral = (Class,{ (ownedAttribute, (enumeration)), (generalization, {NamedElement})}) is a user-defined data value for an enumeration.

- enumeration = (Property,{ (aggregation, shared), (subsettedProperty, {NamedElement::namespace}), (type, Enumeration)}) specifies the ordered set of literals for this enumeration.

Primitive types consist of a set of common mathematical data types, like boolean, integer, and real as well as strings to store text, which have their origin in the computer sciences. The implementation of these types (section 2.5) resulted in several types according to the bytes reserved for each data type e.g., integer is implemented as "normal" integer ranging from -2^{31} to $2^{31} - 1$, up to a type called "BigInteger" that is mainly restricted by the hardware that is used.

7.3.43 PrimitiveType = (Class,{ (generalization, {Datatype})}) defines a predefined data type, without any relevant substructure (i.e., it has no parts). Instances of primitive types do not have identity. If two instances have the same representation, then they are indistinguishable.

17.4.1 Boolean = (Class,{ (generalization, {PackageableElement, TypedElement})}) The boolean type is used for logical expression, consisting of the predefined values true and false.

17.4.2 Integer = (Class,{(generalization, {PackageableElement, TypedElement})}) An instance of Integer is an element of the (theoretically infinite) set of integers {-2, -1, 0, 1, 2}. Possibly platform dependent variants are integer and long (with different, non-infinite platform realizations).

Double = (Class,{(generalization, {PackageableElement, TypedElement})}) An instance of Double is a (theoretically infinite) floating point value. Possibly platform dependent variants are float and double (with different, non-infinite platform realizations).

17.4.3 String = (Class,{ (generalization, {PackageableElement, TypedElement})}) A String is a sequence of characters in a suitable character set used to display information about the model. Character sets may include non-roman alphabets and characters.

In contrast to simple types complex types can introduce a set of simple types/and other complex types.

2.1.2 Complex Types

A complex type consists of additional elements to construct compositions of simple types in form of classes, collection types, and their relationships. Figure 2.4 presents the introduced complex type classes. The definition of complex types is based on the "Class" element that contains a set of attributes. A class introduces a set of attributes. Each attribute specifies the relationship between class and attribute by the AggregationKind. AggregationKind is an Enumeration Instance specified as follows:

7.3.2 AggregationKind = (Enumeration,{ (ownedLiteral, ('none', 'shared', 'composite'))}) is an enumeration type that specifies the literals for defining the kind of aggregation of a property.

- 'none': indicates that the property has no aggregation *(association)*.
- 'shared': indicates that the property has a shared aggregation *(composition target)*.
- 'composite': indicates that the property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts) *(composition source)*.

Class is the core element to define complex types. A Class holds owned attributes and operations and has a reference to its superclass.

7.3.7 Class = (Class,{ (ownedAttribute, (ownedAttribute, ownedOperation, superClass)), (generalization, {Classifier})}) describes a set of objects that share the same specifications of features, constraints, and semantics.

- ownedAttribute = (Property,{ (aggregation, composite), (subsettedProperty, {Classifier::a,Namespace::ownedMember}), (type, Property), (isOrdered, *true*), (lowerValue, 0), (upperValue, *)}) specifies the attributes owned by the class.
- ownedOperation = (Property,{(aggregation, composite), (subsettedProperty, {Classifier::feature,Namespace::ownedMember}), (type, Operation), (isOrdered, *true*), (lowerValue, 0), (upperValue, *)}) specifies the operations owned by the class.
- superClass = (Property,{ (type, Class), (lowerValue, 0), (upperValue, *)}) specifies the superclasses of a class.

A "Classifier" is a super class of a Class. A Classifier introduces additional attributes to specify for example if it is possible to instantiate the class or not, the list of not contained attributes, or the containing package.

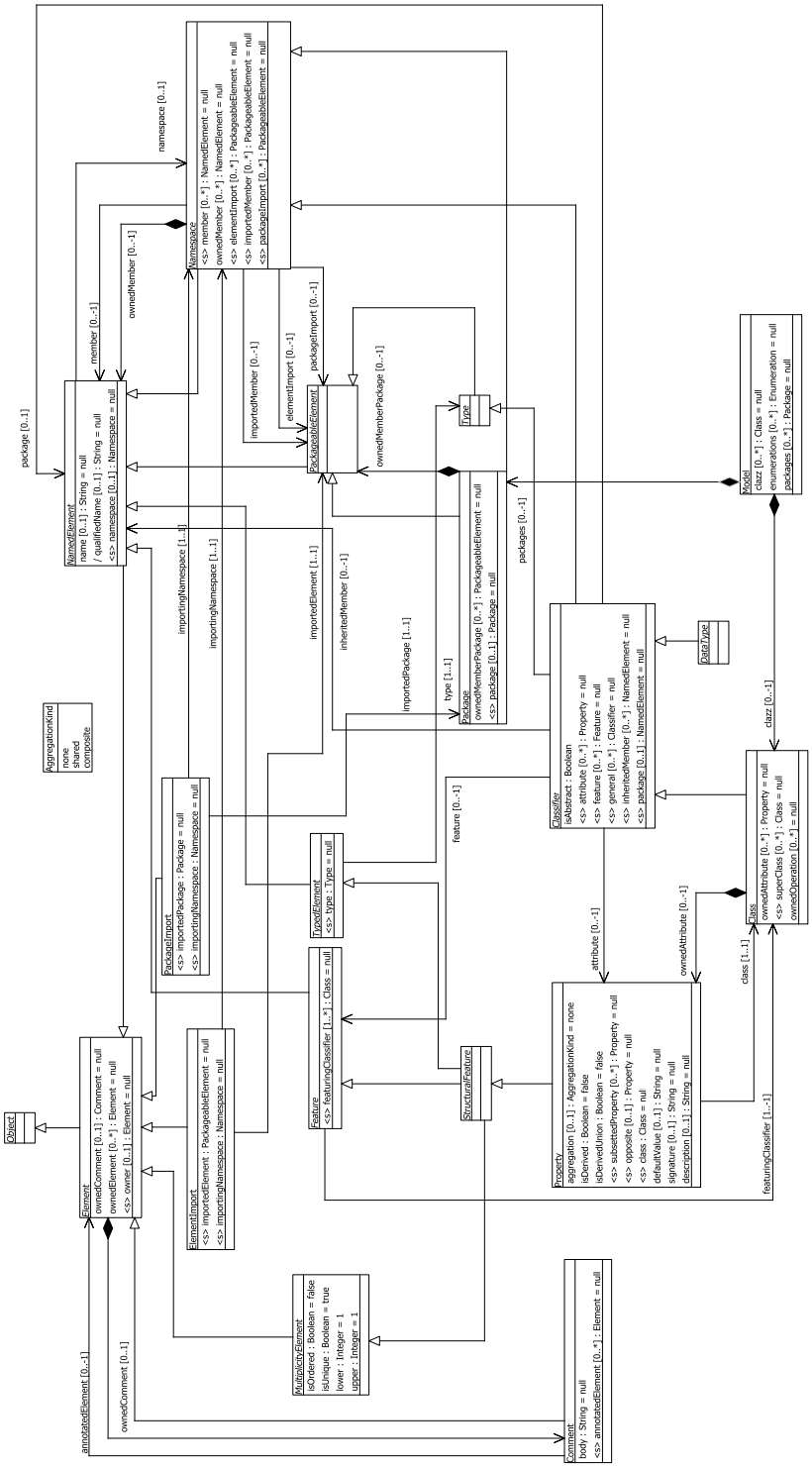


Figure 2.4: Class diagram of complex types

7.3.8 *Classifier* = (Class,{(ownedAttribute, (isAbstract, attribute, feature, general, inheritedmember, package)), (isAbstract, true), (generalization, {Namespace, Type})}) is a name space whose members can include features.

- isAbstract = (Property,{(defaultValue, *false*), (type, Boolean)}) specifies if the classifier does provide a complete declaration and can be instantiated.
- attribute = (Property,{(isDerivedUnion, *true*), (subsettedProperty, {Classifier::feature}), (type, Property), (lowerValue, 0), (upperValue, *)}) refers to all of the properties that are direct (i.e., not inherited or imported) attributes of the classifier.
- feature = (Property,{(isDerivedUnion, *true*), (subsettedProperty, {Namespace::member}), (opposite, Feature::featuredClassifier), (type, Feature), (lowerValue, 0), (upperValue, *)}) specifies each feature defined in the classifier.
- general = (Property,{(type, Classifier), (lowerValue, 0), (upperValue, *)}) specifies the general classifiers for this classifier.
- inheritedMember = (Property,{(isDerived, gen(this) - see Def. 22), (subsettedProperty, {Namespace::member}), (type, NamedElement), (lowerValue, 0), (upperValue, *)}) specifies all elements inherited by this classifier from the general classifiers.
- package = (Property,{(subsettedProperty, {NamedElement::namespace}), (type, NamedElement), (lowerValue, 0)}) specifies the owning package of this classifier, if any.

7.3.11 *DataType* = (Class,{(isAbstract, *true*), (generalization, {Classifier})}) is a type whose instances are identified only by their value. DataType may contain attributes to support the modeling of structured data types.

"MultiplicityElement" defines the container attributes (e.g., the container is ordered) for a set of instances and values.

7.3.32 *MultiplicityElement* = (Class,{(ownedAttribute, (isOrdered, isUnique, ..., upperValue)), (isAbstract, *true*), (generalization, {Element})}) is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A multiplicity element embeds this information to specify the allowable cardinalities for an instantiation of this element.

- isOrdered = (Property,{(defaultValue, *false*), (type, Boolean)}) specifies whether the values in an instantiation of this element are sequentially ordered.

- isUnique = (Property,{ (defaultValue, *true*), (type, Boolean)}) specifies whether the values in an instantiation of this element are unique.
- lowerValue = (Property,{ (subsettedProperty, {Element::ownedElement}), (type, ValueSpecification)}) specifies the lower bound for this multiplicity.
- upperValue = (Property,{ (subsettedProperty, {Element::ownedElement}), (type, ValueSpecification)}) specifies the upper bound for this multiplicity.

"Features" and "Properties" are used to aggregate type definitions. Property attributes are, e.g., derivations to subset/superset other type definitions, default value, and bidirectional opposite properties.

7.3.19 *Feature* = (Class,{ (ownedAttribute, (isStatic, featuringClassifier)), (isAbstract, *true*), (opposite, Classifier::feature), (generalization, {NamedElement})}) declares a behavioral or structural characteristic of instances of classifiers.

- featuringClassifier = (Property,{ (isDerivedUnion, *true*), (type, Classifier), (upperValue, *)}) specifies classifiers that have this Feature as a feature.

7.3.44 Property = (Class,{ (ownedAttribute, (aggregation, isDerived, isDerivedUnion, defaultValue, subsettedProperty, opposite, class)), (generalization, {Structuralfeature})}) is a structural feature. A Property related to a classifier by ownedAttribute represents an attribute.

- aggregation = (Property,{ (defaultValue, none), (type, Aggregationkind)}) specifies the kind of aggregation that applies to the property.
- isDerived = (Property,{ (aggregation, composite), (type, Operation), (lowerValue, 0)}) specifies whether the property is derived (i.e., whether its value or values can be computed from other information and it is derived).
- isDerivedUnion = (Property,{(defaultValue, *false*), (type, Boolean),}) specifies whether the property is derived as the union of all of the properties that are constrained to subset it.
- defaultValue = (Property,{(aggregation, composite), (type, ValueSpecification), (lowerValue, 0)}) is evaluated to give a default value for the property, when an object of the owning classifier is instantiated.
- subsettedProperty = (Property,{ (type, Property), (lowerValue, 0), (upperValue, *)}) references the properties of which this property is constrained to be a subset.
- opposite = (Property,{ (type, Property), (lowerValue, 0)}) gives the other end of a binary relationship in the case where the property is navigable.

- class = (Property,{(aggregation, shared), (type, Class),}) specifies the class of the property.

7.3.49 *StructuralFeature* = (Class,{(isAbstract, *true*), (generalization, {Feature, MultiplicityElement, TypedElement})}) is a typed feature of a classifier that specifies the structure of instances of the classifier.

Types are assigned via the abstract class "TypedElement" and summarized in the meta class "Type".

7.3.51 *Type* = (Class,{ (isAbstract, *true*), (generalization, {PackageableElement})}) serves as a constraint on the range of values represented by a typed element.

7.3.52 *TypedElement* = (Class,{(ownedAttribute, (type)), (isAbstract, *true*), (generalization, {NamedElement})}) A TypedElement is an Element, which has a type that serves as a constraint on the range of values the Element can represent.

- type = (Property,{(type, Type)}) specifies the type of the Element.

Type definitions are followed by the definition of instances to clarify the (valid) instantiation of the type domains.

2.1.3 Instances

This section introduces "Specification" and "Slots" as basic mechanism to instantiate objects. Specification refers the instantiated classifiers and a set of Slots holding the instance values. Each Slot refers the feature defining the "type" of the Slot and refers back to the Specification.

7.3.22 InstanceSpecification = (Class,{ (ownedAttribute, (classifier, slot)), (generalization, {PackageableElement})}) specifies existence of an entity in a modeled system and completely or partially describes the entity.

- classifier = (Property,{ (type, Classifier),}) or classifiers of the represented instance. If multiple classifiers are specified, the instance is classified by all of them.
- slot = (Property,{(aggregation, shared), (subsettedProperty, {Element::ownedElement}), (type, Slot), (lowerValue, 0), (upperValue, *)}) giving the value or values of a structural feature of the instance.

7.3.48 Slot = (Class,{(ownedAttribute, (definingFeature, owningInstance, value)), (generalization, {Element})}) specifies that an entity modeled by an instance specification has a value or values for a specific structural feature.

- definingFeature = (Property,{(type, Structuralfeature)}) specifies the values that may be held by the slot.
- owningInstance = (Property,{(aggregation, composite), (type, Instancespecification), (subsettedProperty, {Element::owner}), (upperValue, *)}) specifies the instance specification that owns this slot.
- value = (Property,{ (aggregation, composite), (subsettedProperty, {Element::ownedElement}), (type, String)}) corresponding to the defining feature for the owning instance specification (e.g., "{first, second, third}").

Besides the object data there exist additional information storage for comments and so called tags.

2.1.4 Information

Additional information for each element is stored in special elements, which are introduced in this section. All elements can be annotated with “Comments” and “Tags”.

7.3.9 Comment = (Class,{(ownedAttribute, (body, annotatedElement)), (generalization, {Element})}) gives the ability to attach various remarks to elements.

- body = (Property,{(type, String)}) specifies a string that is the comment.
- annotatedElement = (Property,{(type, Element), (lowerValue, 0), (upperValue, *)}) references the Element(s) being commented.

Tags are pairs of arbitrary names and values that can be assigned to all elements.

10.1(MOF) Tag = (Class,{(ownedAttribute, (name, value, elements)), (generalization, {Object})}) represents a single piece of information that can be associated with any number of model elements. A model element can be associated with many tags, and the same tag can be associated with many model elements.

- name = (Property,{(type, String)}) the name to distinguish Tags associated with a model element.
- value = (Property,{(type, String)}) the value of the Tag. MOF places no meaning on these values.
- elements = (Property,{(type, Element), (lowerValue, 0), (upperValue, *)}) the value of the Tag. MOF places no meaning on these values.

Tags are often used to extend semantics in a “light way”, e.g., the implementation generates different source code for classes with certain tags. Tagged Values, Constraints and Stereotypes are used to build so called “UML Profiles” to extend or restrict the original syntax in a conservative way.

2.1.5 Structures

This section introduces abstract structure related elements and mechanisms to import a set of elements into a different name space.

9.3(MOF) Object = (Class,{ (isAbstract, true)}) Object is introduced as super-type of Element in order to be able to have a type that represents both elements and data values. Objects represent 'any' value.

7.3.14 Element = (Class,{(ownedAttribute, (ownedComment, ownedElement, owner)), (isAbstract, *true*), (generalization, {Object})}) is a constituent of a Model. As such, it has the capability of owning other elements.

- ownedComment = (Property,{(aggregation, composite), (subsettedProperty, {Element::ownedMember}), (type, Comment), (lowerValue, 0), (upperValue, *)}) specifies the comments owned by this element.
- ownedElement = (Property,{ (aggregation, composite), (isDerivedUnion, *true*), (type, Element), (lowerValue, 0), (upperValue, *)}) specifies the elements owned by this element.
- owner = (Property,{ (aggregation, shared), (isDerivedUnion, *true*), (type, Element), (lowerValue, 0)}) specifies the element owning this element.

Imported elements can be accesses without a namespace reference.

7.3.15 ElementImport = (Class,{(ownedAttribute, (importedElement, importingNamespace)), (generalization, {Element})}) An element import adds the name of a packageable element from a package to the importing namespace.

- importedElement = (Property,{(aggregation, none), (type, PackageableElement), (lowerValue, 1), (upperValue, 1)}) specifies the PackageableElement whose name is to be added to the namespace.
- importingNamespace = (Property,{ (aggregation, none), (type, Namespace), (lowerValue, 1), (upperValue, 1)}) specifies the namespace that imports PackageableElements from other packages.

2.1.6 Naming

The last section of the data structures covers naming related classes regarding definition (named element), structuring of named elements (in form of packages), and importing of named elements.

7.3.33 *NamedElement* = (Class,{(ownedAttribute, (name, qualifiedName, namespace)), (isAbstract, *true*), (generalization, {Element})}) is an element in a model that may have a name.

- name = (Property,{(type, String)}) of the NamedElement.
- qualifiedName = (Property,{ (isDerived, derivedValueQualifiedName - see figure 2.11), (type, String)}) allows the NamedElement to be identified within a hierarchy of nested namespaces.
- namespace = (Property,{ (aggregation, shared), (subsettedProperty, {Element::owner}), (type, Namespace) (lowerValue, 0)}) specifies the namespace that owns the NamedElement.

7.3.34 *Namespace* = (Class,{(ownedAttribute, (elementImport, importedMember, member, ownedMember, ownedRole, packageImport)), (isAbstract, true), (generalization, {NamedElement})}) is an element in a model that contains a set of named elements that can be identified by name.

- elementImport = (Property,{(aggregation, composite), (subsettedProperty, {Element::ownedElement}), (type, PackageableElement), (lowerValue, 0), (upperValue, *)}) references the ElementImports owned by the namespace.
- importedMember = (Property,{(subsettedProperty, {Namespace::member}), (type, PackageableElement), (isDerived, derivedValueImportedMember - see figure 2.12), (lowerValue, 0), (upperValue, *)}) references the PackageableElements that are members of this namespace as a result of either PackageImports or ElementImports.
- member = (Property,{(isDerivedUnion, *true*), (type, NamedElement), (lowerValue, 0), (upperValue, *)}) specifies a collection of NamedElements identifiable within the namespace, either by being owned or by being introduced by importing or inheritance.
- ownedMember = (Property,{ (aggregation, composite), (isDerivedUnion, *true*), (type, NamedElement), (subsettedProperty, {Element::ownedElement, Namespace::member}), (lowerValue, 0), (upperValue, *)}) specifies a collection of NamedElements owned by the namespace.
- ownedRule = (Property,{(aggregation, composite), (isDerivedUnion, *true*), (type, Constraint) (lowerValue, 0), (upperValue, *)}) specifies a set of constraints owned by this namespace.

- packageImport = (Property,{(aggregation, composite), (subsettedProperty, {Element::ownedElement}), (type, PackageableElement), (lowerValue, 0), (upperValue, *)}) references the PackageImports owned by the namespace.

Package group and bind elements to a namespace. Access from other packages can be done via the "." operator for Sup-Packages and ".." for Super-Packages.

7.3.37 Package = (Class,{(ownedAttribute, (ownedMember, package)), (generalization, {Namespace, PackageableElement})}) A package is used to group elements and provides a namespace for the grouped elements.

- ownedMember = (Property,{ (aggregation, composite), (subsettedProperty, {Namespace::ownedElement}), (type, PackageableElement), (lowerValue, 0), (upperValue, *)}) specifies the members owned by the package.
- package = (Property,{ (aggregation, shared), (type, Package), (lowerValue, 0), (upperValue, 1)}) references the owning package of a package.

7.3.38 *PackageableElement* = (Class,{ (isAbstract, *true*), (generalization, {NamedElement})}) indicates a named element that may be owned directly by a package.

7.3.39 PackageImport = (Class,{(ownedAttribute, (importedPackage, importingNamespace)), (generalization, {Element})}) A package import is a relationship between an importing namespace and a package, indicating that the importing namespace adds the names of the members of the package to its own namespace. Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import.

- importedPackage = (Property,{ (aggregation, none), (type, Package), (lowerValue, 1), (upperValue, 1)}) Specifies the package whose members are imported into a namespace.
- importingNamespace = (Property,{(aggregation, none), (type, Namespace), (lowerValue, 1), (upperValue, 1)}) Specifies the namespace that imports a PackageableElement from another package.

The static aspects of RMOF were introduced in the previous pages. The behavioral elements will follow in the next sections.

2.1.7 Core Meta Model and Configurations

Each language layer can contain arbitrary variables that are instantiated during a simulation run fired by the transitions of the symbolic transition system STS. It is

required to define variable snapshots of such a language system means, which is done in the following definitions. The Core Meta Model (based on the previously introduced and enriched class descriptions of the UML and MOF specifications) is defined in this section.

Definition 11 (Core Meta Model $\overset{\circlearrowleft}{m}$) *The core meta model $\overset{\circlearrowleft}{m}$ is defined on base of the introduced EBNF. Let ci $\in$ <Model.ClassInst> be the class instance description of the RMOF data structures. $\overset{\circlearrowleft}{m}$ =($\overset{\circlearrowleft}{m}.n$, $\overset{\circlearrowleft}{m}.n$, $\overset{\circlearrowleft}{m}.I$, $\overset{\circlearrowleft}{m}.T$, $\overset{\circlearrowleft}{m}.A$, $\overset{\circlearrowleft}{m}.O$) is defined as follows:*

- *All classes are build as follows $\forall i = (i.n, \widehat{i}.n, i.A) \in \overset{\circlearrowleft}{m}.I$:*
  - *class name setting: i.n= <ci.InstanceName>.*
  - *meta class setting: $\widehat{i}.n$= <ci.InstanceValues.Identifier>.*
  - *attribute value setting: i.A= <ci.InstanceValues.Attributes>*
- *The set of basic types corresponds to the used types of the data structures, $\overset{\circlearrowleft}{m}.T = \{\mathbb{B}, \mathbb{Z}, S^*\}$, with $S = \{a, \ldots, Z\}$.*
- *The set of pairs of type assignments is $\overset{\circlearrowleft}{m}.A = \{(Kernel :: Boolean, \mathbb{B}), (Kernel :: Integer, \mathbb{Z}), (Kernel :: String, S^*)\}$.*
- *$\overset{\circlearrowleft}{m}.O$ is the set of (primitive) operations, starting at definition 14, ending at definition 22.*

■

All specified instance descriptions are complemented according to their meta instances. A global step executes a single effect on all active State Charts. This is done as long as at least a single active State Chart is available. Each effect is assigned to a transition. A transition is selected depending on a (positive) guard evaluation non-deterministically, if several possible transitions can be executed. Effects can invoke predefined simple operations.

Let *ci* $\in$ <*Model.ClassInst*> be a class instance description of this section and $\overset{\circlearrowleft}{m}$ is the corresponding core meta model. The following abbreviations for the access elements of $\overset{\circlearrowleft}{m}$ are used:

- Q is the set of states of $\overset{\circlearrowleft}{m}$, where <*ci.InstanceValues.Identifier*> = State. There is one State Chart in $\overset{\circlearrowleft}{m}$.

- $q_0 \in Q$, where q_0 is the initial state = $<ci.InstanceValues.Identifier>$ = PseudoState.
- $q_x \in Q$, where q_x is the end state = $<ci.InstanceValues.Identifier>$ and Identifier = FinalState.
- $tr \subseteq Q \times (\{\gamma \mid \gamma \text{ is a guard or an effect}\}) \times c.Q$ is the transition relation.

$$\begin{aligned}\forall(q_1,\gamma,q_2)\ \exists ci \in <Model.ClassInst> :\\ q_1 = <ci.InstanceValues.Attributes.source>\\ \wedge\, q_2 = <ci.InstanceValues.Attributes.target>\\ \wedge\, \gamma = <ci.InstanceValues.Attributes.guard> \wedge guard \neq ""\\ \vee\, \gamma = <ci.InstanceValues.Attributes.effect>\end{aligned}$$

For each type $T_c \in T_C$, O_c denotes the corresponding semantic type and $\mathcal{D}_{O_c} := c \times \mathbb{N}$ defines the domain. O_C with domain $\bigcup_{c \in C} \mathcal{D}_{O_c}$ is called the object reference domain.

A snapshot of all variables of a language layer is called Layer Configuration.

Definition 12 (Layer Configuration) *Let $\sum \overset{\circlearrowleft}{M}$ be the set of all $\overset{\circlearrowleft}{M}$ layers, $l = (l.n, \widehat{l}.n, l.I, l.T, l.A, l.O) \in \sum \overset{\circlearrowleft}{M}$. A layer configuration consists of an attribute configuration 'ac' of type*

$$\mathcal{T}_{ac} := \bigcup_{i=(i.n,\widehat{i}.n,i.A)\in l.I,\ a=(a.n,a.v)\in i.A} (a.n \to \mathcal{T}_{T(l)})$$

The type of the layer configuration of l is $\mathcal{T}_{lc}(l) := \mathcal{T}_{ac}$ ■

A System Configuration is a snapshot of all variables of all layers the system consists of.

Definition 13 (System Configuration) *Let $\sum \overset{\circlearrowleft}{M}$ be the set of all $\overset{\circlearrowleft}{M}$ layers, $\overset{\circlearrowleft}{M}$ be a system, l be a layer of $\overset{\circlearrowleft}{M}$. The system configuration is defined as:*

$$\mathcal{T}_{sconf}(\overset{\circlearrowleft}{M}) := \bigcup_{l \in \overset{\circlearrowleft}{M}} \mathcal{T}_{lc}(l)$$

The next section introduces expressions to change and query configurations.

2.2 Expressions

The system can execute an action or pass a guard if the current transition (q, γ, q') is enabled (meaning the source state is active) and annotated with one of the following: The predicate $guard : \sum \overset{\circlearrowleft}{M} \times Expr \to \mathbb{B}$ processes guards on transitions:

$$guard_{\overset{\circlearrowleft}{M}}(\text{“}[expr]''') := expr = true \wedge sysfail' := (sysfail \vee expr = \bot).$$

The predicate $assign : \sum \overset{\circlearrowleft}{M} \times Expr \to \mathbb{B}$ assigns a new value to an instance:

$$assign_{\overset{\circlearrowleft}{M}}(\text{“}\tau.a := expr'') := expr \neq sysfail \Rightarrow \tau.a' := expr \\ \wedge sysfail' := (sysfail \vee expr = \bot).$$

A sequence of assignments can be defined by using the sequence operator “;”. This section introduces the expressions available in RMOF. Starting with a representation of relevant classes and refined in terms of formal expression syntax and semantics.

7.3.54 *ValueSpecification* = (Class,{(ownedAttribute, (expression)), (generalization, {PackageableElement, TypedElement})}) specifies a (possibly empty) set of instances, including both objects and data values.

- expression = (Property,{ (subsettedProperty, {Element::owner}), (type, String), (lowerValue, 0)}) specifies the owning expression if this value specification is an operand.

The OMG specifications avoid a (detailed) expression syntax definition. This work elaborates the expression language, starting by the differentiation of guards and actions.

a.2 GuardValueSpecification = (Class,{(ownedAttribute, (expression)), (generalization, {PackageableElement, TypedElement})}) specifies a (possibly empty) set of instances, including both objects and data values.

- expression = (Property,{(subsettedProperty, {Element::owner}), (type, GuardExpression), (lowerValue, 0)}) specifies the owning expression if this value specification is an operand.

a.3 ActionLanguageOperation = (Class,{(ownedAttribute, (symbol, operand)), (generalization, {ValueSpecification})}) is a structured tree of primitive operations that denotes a (possibly empty) set of values when evaluated in a context.

- symbol = (Property,{(type, PrimitiveOperation)}) specifies the PrimitiveOperation associated with the node in the expression tree.

This section is concluded with primitive operations. Primitive operations are defined as mathematical functions (and should be supported by the platform that implements RMOF). In contrast to complex operations, which are defined in terms of behavior models, i.e., state machines of operations.

Definition 14 (Binary Arithmetic Operations) *The arithmetic operations with two operands comprise* $\{+, -, *, /, \div, mod\}$*, (*$\div$ *is a whole number division) where, e.g,* $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$*, with*

$$+(a,b) := \begin{cases} \bot & \text{if } a \in \{\bot, \mathsf{nil}\} \vee b \in \{\bot, \mathsf{nil}\} \\ a+b & else \end{cases}$$

Analog defined for the rest of the operations and the domains $\mathbb{N}, \mathbb{Z}, \mathbb{R}$*. The operation is written in infix notation and* $a+ = 1$ *is syntactic sugar for* $a = a + 1$*.*

Definition 15 (Unary Arithmetic Operations) *The arithmetic operations with a single operand comprise* $\{++, --, !\}$*, where e.g,* $! : \mathbb{B} \to \mathbb{B}$*, with*

$$!(a) := \begin{cases} \bot & \text{if } a \in \{\bot, \mathsf{nil}\} \\ true & \text{if } a \in \{true\} \\ false & else \end{cases}$$

Whereby $a++$ *and* $b--$ *are shortcuts for* $a = a + 1$ *respectively* $b = b - 1$*.*

Definition 16 (Comparison Operations) *The comparison operations comprise* $\{==, !=, <=, >=\}$*, where e.g,* $==: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$*, with*

$$==(a,b) := \begin{cases} \bot & \text{if } a \in \{\bot, \mathsf{nil}\} \vee b \in \{\bot, \mathsf{nil}\} \\ true & a = b \\ false & else \end{cases}$$

Analog defined for the rest of the operations and the for domains $\mathbb{N}, \mathbb{Z}, \mathbb{R}$*. The operation is written in infix notation.*

Definition 17 (Type Conversion Operations) *The conversion operations comprise* $\{toBoolean, toInteger, toDouble, toString\}$*, where e.g,* $toInteger : \mathbb{S}^* \times \to \mathbb{N}_0$*, with*

$$toInteger(a) := \begin{cases} n & \exists n \in \mathbb{N}_0 \text{ with } a == "n" \\ \bot & else \end{cases}$$

Analog defined for all primitive types $\mathbb{N}, \mathbb{Z}, \mathbb{R}, S^*$ *(if there exists at least one instance of the first type that can be conversed into the the second type.* ■

Primitive operations cover also so called "Multi-Valued Types" like sets, bags and their ordered variants ordered sets and sequences, including membership relationships, sub setting, and conversion operations. The permutation set of a given set including error values is defined as $e := 2^{s^*} \cup \{\bot, \mathsf{nil}\}$, with $s = \{a, b, \ldots, Z, 0, \ldots, 9\}$ as "constructing syntax core".

Definition 18 (Set Operations) *Let $\sum S$ be the set of all sets of elements, tuples, sets, ordered sets, bags, and sequences that may be build (reflexively) of e, $\sum E \subseteq \sum S$, $E \in \sum S$. The set functions comprise at least:*

- *"flat" is element of* $contains_s : \sum S \times E \to \mathbb{B}$ *is defined*

$$contains_s(S, e) := \begin{cases} true & if\ e \in S \\ & \wedge \neg containsOne_s^d(add_s(S, e), \{\bot, \mathsf{nil}\}) \\ false & if\ e \notin S \\ & \wedge \neg containsOne_s^d(add_s(S, e), \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *"deep" is element of* $contains_s^d : \sum S \times E \to \mathbb{B}$ *is defined*

$$contains_s^d(S, e) := \begin{cases} true & if\ contains(S, e) \vee \exists s \in S : \\ & contains_s^d(s, e) \\ false & if\ \neg contains(S, e) \wedge \nexists s \in S : \\ & contains_s^d(s, e)) \\ \bot & else \end{cases}$$

- *"flat" at least one is element of* $containsOne_s : \sum S \times \sum E \to \mathbb{B}$ *is defined*

$$containsOne_s(S, E) := \begin{cases} true & if\ \exists e \in E : contains_s(S, e) \\ false & if\ \nexists e \in E : contains_s(S, e) \\ \bot & else \end{cases}$$

- *"deep" at least one is element of* $containsOne_s^d : \sum S \times \sum E \to \mathbb{B}$ *is defined*

$$containsOne_s^d(S, E) := \begin{cases} true & if\ containsOne(S, E) \vee \exists s \in S : \\ & containsOne_s^d(s, e) \\ false & if\ \neg containsOne(S, E) \wedge \nexists s \in S : \\ & containsOne_s^d(s, e) \\ \bot & else \end{cases}$$

- *"flat" subset of subset* $subset_s : \sum S \times \sum E \to \mathbb{B}$ *is defined*

$$subset_s(S,E) := \begin{cases} true & if\ \forall e \in E : contains_s(S,e) \\ false & if\ \exists e \in E : \neg contains_s(S,e) \\ \bot & else \end{cases}$$

- *"deep" subset of subset* $subset_s^d : \sum S \times \sum E \to \mathbb{B}$ *is defined*

$$subset_s^d(S,E) := \begin{cases} true & if\ \forall e \in E : contains_s(S,e) \vee \exists e \in E : \\ & isIn_s^d(S,e) \\ false & if\ \exists e \in E : \neg contains_s(S,e) \wedge \forall e \in E : \\ & negisIn_s^d(S,e) \\ \bot & else \end{cases}$$

- *join set and element* $add_s : \sum S \times E \to \sum S$ *is defined*

$$add_s(S,e) := \begin{cases} S \cup \{e\} & if\ \neg containsOne_s^d(add_s(S,e), \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *join two sets* $add_s : \sum S \times \sum S \to \sum S$ *is defined*

$$add_s(S,S') := \begin{cases} add_s(\ldots(add_s(S,s_1),\ldots),s_n) & if\ S' = \{s_1,\ldots,s_n\} \\ \bot & else \end{cases}$$

- *remove an element from a set* $rm_s : \sum S \times E \to \sum S$ *is defined*

$$rm_s(S,e) := \begin{cases} S \setminus \{e\} & if\ \neg containsOne_s^d(add_s(S,e), \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *remove a set from a set* $rm_s : \sum S \times \sum S \to \sum S$ *is defined*

$$rm_s(S,S') := \begin{cases} rm_s(\ldots(rm_s(S,s_1),\ldots),s_n) & if\ S' = \{s_1,\ldots,s_n\} \\ \bot & else \end{cases}$$

- *count all set elements* $size_s : \sum S \to \mathbb{N}_0$ *is defined*

$$size_s(S) := \begin{cases} |s| & if\ \neg containsOne_s^d(add_s(S,e), \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *remove all elements from a set* $clear_s : \sum S \to \sum S$ *is defined:*
  $clear_s(S) := \{\}$

■

In contrast to sets an ordered set is totally ordered. Since the operations for sets also apply for ordered sets mapping operations between sets and ordered sets are defined instead of redefining all set operations for ordered sets. Additional operations are required to obey the ordering of an ordered set ,e.g., when introducing new elements to the ordered set.

Definition 19 (Ordered Set Operations) *An ordered set is a set, that is totally ordered, written* $[o_1, \ldots, o_n]$. *Let* $\sum O$ *be the set of all ordered sets of all elements, tuples, sets, ordered sets, bags, and sequences that may be build (reflexively) of e,* $\sum E \subseteq \sum O$, $E \in \sum O$. *The variables* $n, i \in \mathbb{N}_0$. *The ordered set functions comprise at least:*

- *make a set of an ordered set* $\overset{os}{\underset{s}{\rightarrow}}: \sum O \rightarrow \sum S$ *is defined:*

$$\overset{os}{\underset{s}{\rightarrow}}(O) := \{o_1, \ldots, o_n\} \text{ with } O = [o_1, \ldots, o_n]$$

  All defined set functions are accessible by previously using $\overset{os}{\underset{s}{\rightarrow}}$ *to transform the ordered set into a set.* $count_o(O)$ *is written for* $count_s(\overset{os}{\underset{s}{\rightarrow}}(O))$. *If the set function returns a set as result a corresponding ordered set including a random ordering. Therefore making an ordered set of a set* $\overset{s}{\underset{os}{\rightarrow}}: \sum S \rightarrow \sum O$ *is defined:*

$$\overset{s}{\underset{os}{\rightarrow}}(S) := [o_1, \ldots, o_n] \text{ with } S = \{o_1, \ldots, o_n\}$$

  If functions change (e.g., $count_o$*) is "overwritten", the new definition is used.*

- *make an ordered set of a set* $\overset{s}{\underset{os}{\rightarrow}}: \sum S \rightarrow \sum O$ *is defined:*

$$\overset{s}{\underset{os}{\rightarrow}}(S) := [o_1, \ldots, o_n] \text{ with } S = \{o_1, \ldots, o_n\}$$

  The ordering is arbitrarily chosen, but after the operation application there exists one.

- *get position of an element of an ordered set* $getPos_o : \sum O \times E \rightarrow \mathbb{N}_0$ *is defined:*

$$getPos_o(O, o_i) := \begin{cases} i & if\ isIn_o(O, o_i) \wedge O = [o_1, \ldots, o_i, \ldots, o_n] \\ \mathsf{nil} & if\ \neg isIn_o(O, o_i) \\ \bot & else \end{cases}$$

- *get element on a position of an ordered set* $getElem_o : \sum O \times \mathbb{N}_0 \rightarrow E$ *is defined:*

$$getElem_o(O, i) := \begin{cases} o_i & if\ isIn_o(O, o_i) \wedge O = [o_1, \ldots, o_i, \ldots, o_n] \\ \mathsf{nil} & if\ size_o(O) = 0 \vee i \notin \{o_1, \ldots, size_o(O)\} \\ \bot & else \end{cases}$$

- *add an element at a certain position in an ordered set* $addAt_o : \sum O \times \mathbb{N} \times E \rightarrow \sum O$ *is defined:* $addAt_o(O, i, e) :=$

$$\begin{cases} [o_1, \ldots, o_{i-1}, e, o_i, \ldots, o_n] & if\ O = [o_1, \ldots, o_n] \wedge \neg isIn_o(O, e) \\ \mathsf{nil} & if\ isIn_o(O, e) \vee i > size_o(O) + 2 \\ \bot & else \end{cases}$$

- *remove an element at a certain position in an ordered set* $rmAt_o : \sum O \times \mathbb{N} \to \sum O$ *is defined:* $rmAt_o(O, i) :=$

$$\begin{cases} [o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n] & if\ O = [o_1, \ldots, o_i, \ldots, o_n] \\ & \wedge getPos(O, i) = o_i \\ getPos_o(O, i) & else \end{cases}$$

■

In the following a so-called "bag" as folded variant of a multi-set is introduced. A multi-set is a set with multiple occurrences of one element. A bag folds the occurrences by introducing pairs of elements with natural numbers, whereby the natural number corresponds to the number of occurrences of the element in the multi-set. Since set operations apply also for bags, mapping operations between bags and sets as well as additional operations required to manage bags are introduced.

Definition 20 (Bag) *Let* S *be a set.* $\mathfrak{B}_S = \{(s_1, n_1), \ldots, (s_m, n_m)\} \in 2^{S \times \mathbb{N}}$ *is called bag of* S *iff* $\forall i, j \in \{1, \ldots, m\} : s_i \neq s_j$[1]*. The set of all bags* B *of* S *is defined as* $\sum \mathfrak{B}_S := \{\mathfrak{B}_S \in 2^{S \times \mathbb{N}} | \mathfrak{B}_S \text{ is a bag}\}$*. Each bag* $\mathfrak{B}_S = \{(s_1, n_1), \ldots, (s_m, n_m)\}$ *can also be written as multi-set in the form* $\{\{s_1, \ldots, s_1, \ldots, s_m, \ldots, s_m\}\}$*, whereby each* s_i *occurs* n_i *times in the multi-set. For all bag related sub-definitions of predicates and functions* S, T *are sets and* $\mathfrak{B}_S \in \sum \mathfrak{B}_S, \mathfrak{B}_T \in \sum \mathfrak{B}_T$ *are corresponding elements of their sets of all bags of* S *and* T*. Further is* $s \in S, t \in T, n, m \in \mathbb{N}$*. The bag predicates and functions are defined as follows:*

- *create a set of a bag is not required regarding the folded set variant. Nevertheless it is possible to write* $size_b(\mathfrak{B}_S)$ *instead of* $size_s(\mathfrak{B}_S)$*.*
- *bag element with minimal occurrences* $min_b : \sum \mathfrak{B}_S \times \mathfrak{B}_T \to \mathbb{B}$ *is defined as follows:*

$$minIn_b(\mathfrak{B}_S, (s, m)) := \begin{cases} true & if\ \exists! n \in \mathbb{N} : isIn_b(\mathfrak{B}_S, (s, n)) \wedge m \leq n \\ false & if\ \exists! n \in \mathbb{N} : isIn_b(\mathfrak{B}_S, (s, n)) \wedge m > n \\ \bot & else \end{cases}$$

- *subset with minimal occurrences* $subset_b : \sum \mathfrak{B}_S \times \sum \mathfrak{B}_T \to \mathbb{B}$ *is defined as follows:*

$$subset_b(\mathfrak{B}_S, \mathfrak{B}_T) := \begin{cases} true & if\ \forall \mathfrak{b} \in \mathfrak{B}_S : min_b(\mathfrak{B}_T, \mathfrak{b}) \\ false & if\ \exists \mathfrak{b} \in \mathfrak{B}_S : \neg min_b(\mathfrak{B}_T, \mathfrak{b})) \\ \bot & else \end{cases}$$

[1] Especially S can be a Cartesian product of other sets.

- *adding a bag element to a bag* $add_b : (\sum \mathfrak{B}_S \times \mathfrak{B}_T) \to \sum \mathfrak{B}_{S \cup \{t\}}$ *is defined as follows:* $add(\mathfrak{B}_S, (s,m)) :=$

$$\begin{cases} add_b(rm_b(\mathfrak{B}_S,(s,n)),(s,m+n)) & if\ \ \exists! n \in \mathbb{N} : isIn_b(\mathfrak{B}_S(s,n)) \\ add_b(\mathfrak{B}_S,(s,m)) & if\ \ \nexists! n \in \mathbb{N} : isIn_b(\mathfrak{B}_S(s,n)) \\ \bot & else \end{cases}$$

  Adding up two bags is analogously defined as joining to sets (based on add_b*).*

- *subtracting a bag element from a bag* $rm_b : (\sum \mathfrak{B}_S \times \mathfrak{B}_T) \to \sum \mathfrak{B}_{S \cup \{t\}}$ *is defined as follows:* $rm_b(\mathfrak{B}_S, (s,m)) :=$

$$\begin{cases} add_b(rm_b(\mathfrak{B}_S,(s,n)),(s,n-m)) & if\ \ \exists! n \in \mathbb{N} : min_b(\mathfrak{B}_S,(s,n)) \\ \mathsf{nil} & if\ \ \nexists! n \in \mathbb{N} : min_b(\mathfrak{B}_S(s,n)) \\ \bot & else \end{cases}$$

  Subtraction of one bag from another is defined as removing a set from a set (based in rm_b*).*

- *count the number of a specific bag elements of a bag* $sizeElements_b : \sum \mathfrak{B}_S \times E \to \mathbb{N}_0$ *is defined as follows:*

$$sizeElements_b(\mathfrak{B}_S, e) := \begin{cases} i & if\ \exists! i \in \mathbb{N} : isIn_b(\mathfrak{B}_S,(e,i)) \\ 0 & if\ \nexists! i \in \mathbb{N} : isIn_b(\mathfrak{B}_S,(e,i)) \\ \bot & else \end{cases}$$

- *count the number of all bag elements of a bag* $size_b : \sum \mathfrak{B}_S \to \mathbb{N}_0$ *is defined as follows:*

$$size_b(\mathfrak{B}_S) := \begin{cases} \sum\limits_{e \in \mathfrak{B}_S} sizeElements_b(e) & if \\ & \neg containsOne_s^d(\mathfrak{B}_S, \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *remove all elements from a bag* $clear_b : \sum \mathfrak{B}_S \to \sum \mathfrak{B}_S$ *is defined:* $clear_b \mathfrak{B}_S) := \{\{\}\}$

■

Sequences are multi-sets with an ordering in contrast to bags. Therefore mapping operations between bags and sequences as well as additional operations to handle ordering information are introduced.

Definition 21 (Sequence Operations) *Let $S = \{s_1, \ldots, s_n\}$ be a set (constructed like in "Primitive Set Operations"). $S^* = \mathfrak{Q}_S = s_i, \ldots, s_k = [[s_i, \ldots, s_k]]_S$ is called a sequence of S, if all $s_i, s_k \in S$. All sequences of S are written as $\sum \mathfrak{Q}_S$. The variables $n, i \in \mathbb{N}_0$. The sequence functions comprise at least:*

- *make a bag of a sequence* $\overset{q}{\underset{b}{\rightarrow}}: \sum \mathfrak{Q}_S \rightarrow \sum \mathfrak{B}_S$ *is defined:*

$$\overset{q}{\underset{b}{\rightarrow}}(Q) := add_q(\ldots(add_q(\{(q_1, n_1)\}, (q_2, n_2), \ldots, (q_n, m_n))$$
$$\text{with } Q = [[q_1, \ldots, q_n]]$$

  where q_1 occurs n_1 times, ..., q_n occurs m_n times in sequence Q. All defined set functions are accessible by previously using $\overset{q}{\underset{b}{\rightarrow}}$ to transform a sequence into a set. $count_q(Q)$ is written for $count_s(\overset{q}{\underset{b}{\rightarrow}}(Q))$. Functions If functions change (e.g., $count_q$) is "overwritten", the new definition is used instead.

- *get position of an element of a sequence* $getPos_q : \sum \mathfrak{Q}_S \times \mathbb{N} \rightarrow E$, *with*

$$getPos_q(\mathfrak{Q}_S, (s_i)) := \begin{cases} i & if\ \mathfrak{Q}_S = [[s_1, \ldots, s_i, \ldots, s_m]]_S \\ & \quad \wedge \neg containsOne_s^d(add_s(\mathfrak{Q}_S, s_i), \{\bot, \mathsf{nil}\}) \\ \mathsf{nil} & if\ \mathfrak{Q}_S = [[s_1, \ldots, s_m]]_S \wedge i > m \\ & \quad \wedge \neg containsOne_s^d(\mathfrak{Q}_S, \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *get element on a certain position of a sequence* $getElem_q : \sum \mathfrak{Q}_S \times \mathbb{N} \rightarrow E$, *with*

$$getElem_q(\mathfrak{Q}_S, i) := \begin{cases} s_i & if\ \mathfrak{Q}_S = [[s_1, \ldots, s_i, \ldots, s_m]]_S \\ & \quad \wedge \neg containsOne_q^d(add_s(\mathfrak{Q}_S, s_i), \{\bot, \mathsf{nil}\}) \\ \mathsf{nil} & if\ \mathfrak{Q}_S = [[s_1, \ldots, s_m]]_S \wedge i > m \\ & \quad \wedge \neg containsOne_q^d(\mathfrak{Q}_S, \{\bot, \mathsf{nil}\}) \\ \bot & else \end{cases}$$

- *add an element at a certain position in a sequence* $addAt_q : \sum \mathfrak{Q}_S \times \mathbb{N} \times E \rightarrow \sum \mathfrak{Q}_S$ *is defined:* $addAt_q(Q, i, e) :=$

$$\begin{cases} [q_1, \ldots, q_{i-1}, e, q_i, \ldots, q_n] & if\ O = [o_1, \ldots, o_n] \\ & \quad \wedge \neg containsOne_q^d(add_q(\mathfrak{Q}_S, e), \{\bot, \mathsf{nil}\}) \\ \mathsf{nil} & i > count_q(O) + 2 \\ \bot & else \end{cases}$$

If i is not specified, $count_q(Q)$ is used instead. Adding up two sequences is analogously defined as joining to sets
(based on $addAt_q(Q, e, count_q(Q))$).

- *remove an element at a certain position in a sequence* $rmAt_q : \sum \mathfrak{Q}_S \times \mathbb{N} \to \sum \mathfrak{Q}_S$ *is defined:* $rmAt_q(Q, i) :=$

$$\begin{cases} [q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n] & if\ Q = [q_1, \ldots, q_i, \ldots, q_n] \\ & \wedge getPos(Q, i) = q_i \\ getPos_q(Q, i) & else \end{cases}$$

Subtracting two sequences is analogously defined as removing a set from a set (based on $rmAt_q$).

- *remove all elements from a sequence* $clear_q : \sum \mathfrak{Q}_S \to \sum \mathfrak{Q}_S$ *is defined:* $clear_q(Q) := []$

■

The defined operations so far support arithmetics and management of multi-valued types. Since languages can be defined on different language layers operations bridging the gap between two layers, e.g., to read of attributes, change layers, as well as create, remove, and filter instances in order to manage (in particular execute) languages are introduced.

Definition 22 (Layer Operations) *Let $\sum \overset{\circlearrowleft}{M}$ be the set of all $\overset{\circlearrowleft}{M}$ systems, $\overset{\circlearrowleft}{M} \in \sum \overset{\circlearrowleft}{M}$, $\overset{\circlearrowleft}{M}_I$ be the set of all instances of $\overset{\circlearrowleft}{M}$, and $N = \{a, \ldots, Z, 0, \ldots, 9\}^*$ is the ID construction set. The following set of helping operations is defined:*

- $up : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \to N$ *determining the meta ID of an ID is defined as follows:*

$$up_{\overset{\circlearrowleft}{M}, l.n}(n) := \begin{cases} i_2.n & if\ j \in \{1, 2\} \wedge \exists i_j = (i_j.n, \widehat{i_j}.n, i_j.A) \in \overset{\circlearrowleft}{M}_I \\ & \quad \exists l_j = (l_j.n, l_j.\widehat{n}, l_j.I, l_j.T, l_j.A, l_j.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad i_j \in l_j.I \wedge n = i_1.n \wedge \widehat{i_1}.n = i_2.n \\ & \quad\quad \wedge\ l.n = l.n_1 \wedge l_1 \dashrightarrow l_2 \\ \bot & else \end{cases}$$

The parameters $\overset{\circlearrowleft}{M}, l.n$ are written as index indicating that they are normally set by the system. When up is applied on itself (e.g., $up(up(*))$) then each application

of up changes the current layer to its meta layer reference. A slightly different operation is executed when n=''*
(with $up : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times \{*\} \to 2^N$*)*

$$up_{\overset{\circlearrowleft}{M},l.n}(n) := \bigcup_{n \in \{i.n | (i.n, \widehat{i}.n, i.A) \in l.n\}} \{up(n)\}$$

- *down* $: \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times 2^N \to 2^N$ *determining the instance IDs of a set of IDs is defined as follows:*

$$down_{\overset{\circlearrowleft}{M},l.n}(N) := \begin{cases} \bigcup_{n.i \in l.n_i} \{n.i\} & \text{if } \exists l.n_i \in \overset{\circlearrowleft}{M} : l.n_i \dashrightarrow l.n \\ & \quad \wedge \bigcup_{n.i \in l.n_i} \{up(n.i)\} = N \\ \bot & \text{else} \end{cases}$$

When down is applied on itself (e.g., down(up(down(i_1*)))) each application of down changes the current layer to its instance layer reference. A slightly different operation is executed when n='*'*
(with $down : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times \{*\} \to 2^N$*)*

$$down_{\overset{\circlearrowleft}{M},l.n}(n) := down_{\overset{\circlearrowleft}{M},l.n} \bigcup_{(i.n, \widehat{i}.n, i.A) \in l.I \wedge (l.n, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M}} \{i.n\}$$

- $read : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \to N$ *reading an attribute value is defined as follows*

$$read_{\overset{\circlearrowleft}{M},l.n}(i.n, a.n) := \begin{cases} a.v & \text{if } \exists (a.n_2, a.v) \in I.A \ \exists (i.n_2, \widehat{i}.n, i.A) \in l.I \\ & \quad \exists (l.n_2, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad i.n = i.n_2 \wedge l.n = l.n_2 \wedge a.n = a.n_2 \\ \bot & \text{else} \end{cases}$$

- $reads : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times 2^N \times N \to 2^N$ *reading attribute values is defined as follows:*

$$reads_{\overset{\circlearrowleft}{M},l.n}(i.N, a.n) := \bigcup_{i.n \in i.N} read_{\overset{\circlearrowleft}{M},l.n}(i.n, a.n)$$

- $write : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \to \mathbb{B}$ *writing an attribute value is defined as follows*

$$write_{\overset{\circlearrowleft}{M},l.n}(i.n, a.n, a.v) := \begin{cases} true & if\ \exists(a.n_2, a.v) \in I.A\ \exists(i.n_2, \widehat{i}.n, i.A) \in l.I \\ & \quad \exists(l.n_2, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad i.n = i.n_2 \wedge l.n = l.n_2 \wedge a.n = a.n_2 \\ & \quad\quad \wedge\ a.v' := a.v \\ false & else \end{cases}$$

- *Let* $p = \{'\{','\}','\{\{','\}\}','[',']','[[',']]', a, \ldots, Z, 0, \ldots, 9, ' ::'\}^*, P \subseteq 2^{p \times p}$ *filter a set of instances on base of a set of attribute value pairs,*
  $filter : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times 2^N \times 2^P \to 2^N$ *defined as follows:*

$$filter_{\overset{\circlearrowleft}{M},l.n}(N, A) := \{n \in N | \forall (a.n, a.v) \in A : read_{\overset{\circlearrowleft}{M},l.n}(n, a.n) = a.v\}$$

- *Let* $p = \{'\{','\}','\{\{','\}\}','[',']','[[',']]', a, \ldots, Z, 0, \ldots, 9, ' ::'\}^*, P \subseteq 2^{p \times 2^{p \times p}}$ *filter a set of instances on base of a set of attribute value pairs with set values,* $filters : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times 2^N \times 2^P \to 2^N$ *defined as follows:*

$$filters_{\overset{\circlearrowleft}{M},l.n}(N, A) := \{n \in N | \forall (a.n, A.v) \in A\ \exists a.v \in A.v : read_{\overset{\circlearrowleft}{M},l.n}(n, a.n) = a.v\}$$

- $rmI : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \to \mathbb{B}$ *remove an instance is defined as follows:*

$$rmI_{\overset{\circlearrowleft}{M},l.n}(n) := \begin{cases} true & if\ \exists(i.n_2, \widehat{i}.n, i.A) \in l.I \\ & \quad \exists(l_2.n, l_2.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad i.n = i_2.n \wedge l.n = l_2.n \\ & \quad\quad \Rightarrow l.I' := remove_s(l.I, (i.n, \widehat{i}.n, i.A)) \\ false & else \end{cases}$$

- $addI : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \times 2^{N \times N} \to \mathbb{B}$ *add an instance is defined as follows:*

$$addI_{\overset{\circlearrowleft}{M},l.n}(i.n, \widehat{i}.n, i.A) := \begin{cases} true & if\ \neg(\exists(i_2.n, \widehat{i}.n, i.A) \in l.I \\ & \quad \exists(l_2.n, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad i.n = i_2.n \wedge l.n = l_2.n) \\ & \quad\quad \Rightarrow l.I' := add_s(l.I, (i.n, \widehat{i}.n, i.A)) \\ false & else \end{cases}$$

- $rmA : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \to \mathbb{B}$ *remove an attribute is defined as follows:* $rmA_{\overset{\circlearrowleft}{M},l.n}(i.n, a.n) :=$

$$\begin{cases} true & \text{if } (a_2.n, a.v) \in i.A \exists (i_2.n, \widehat{i.n}, i.A) \in l.I \\ & \quad \exists (l_2.n, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad a.n = a_2.n \wedge i.n = i_2.n \wedge l.n = l_2.n \\ & \quad\quad \Rightarrow i.A' := removem_s(i.A, (a.n, a.v)) \\ false & else \end{cases}$$

- $addA : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \times N \to \mathbb{B}$ *add an attribute is defined as follows:* $addA_{\overset{\circlearrowleft}{M},l.n}(i.n, a.n, a.v) :=$

$$\begin{cases} false & \text{if } \neg((a_2.n, a_2.v) \in i.A \exists (i_2.n, \widehat{i.n}, i.A) \in l.I \\ & \quad \exists (l_2.n, l.\widehat{n}, l.I, l.T, l.A, l.O) \in \overset{\circlearrowleft}{M} : \\ & \quad\quad a.n = a_2.n \wedge i.n = i_2.n \wedge l.n = l_2.n) \\ & \quad\quad \Rightarrow i.A' := add_s(i.A, (a.n, a.v)) \\ false & else \end{cases}$$

- *checks if type of an attribute is of a given class* $isTypeOf : \sum \overset{\circlearrowleft}{M} \times \overset{\circlearrowleft}{M} \times N \times N \to \mathbb{B}$*, is defined as follows:* $isTypeOf_{\overset{\circlearrowleft}{M},l.n}(a.n, i.n) :=$

$$\begin{cases} true & \text{if } type(a.n) \in type_{hierarchy}(i.n) \\ false & else \end{cases}$$

  The $type_{hierarchy}$ *is a set containing all super classes of the given class.*

- *up and down can be added as suffix, e.g.,* $createInstance\_down\_down(\ldots)$*, short* $createInstance\_d\_d(\ldots)$*, or* $createInstance\_d2(\ldots)$*. Then the given layer is increased (up) or decreased (down) for the operation.*

■

'.' can be used to access instance attributes instead of read/write. Let $i = (i.n, \widehat{i.n}, i.A)$ with $(a.n, a.v) \in i.A$ then $i.n.a.n \mathrel{\widehat{=}} read(i.n, a.n)$ and $i.n.a.n = 42 \mathrel{\widehat{=}} write(i.n, a.n)$. A synonym for a *size* operator is # which is frequently used in the RMOF models.

The expression definitions are complemented with the interpretation ordering that is applied for all kinds of expression composition.

Definition 23 (Interpretation Ordering) *The interpretation ordering is normally from left to right, e.g., x := y / 0 + 1 would result in an error. The operations have the following ranking regarding their interpretation.*

Operation	*Rank*
$++, --, !$	*1*
$*, /, \div$	*2*
$+, -$	*3*
$==, != , <=, >=$	*5*
Rest of the operations (e.g., add$_s$)	*4*

Brackets can be used to change the ordering, e.g., x := y / (0 + 1). ■

The implemented action language differs sometimes from the specified one due to the platform mapping. This is in particular the case when predefined platform operations exist, e.g., ".size()".

2.3 Dynamics

This section introduces algorithms, meaning the syntax and semantics to describe and execute the previously introduced expressions. Syntax is introduced in form of a state machine in section 2.3.1. Formal semantics is defined as a symbolic transition system in section 2.3.2. Syntax and semantics are reflexively defined. This concludes the first step of the semantic definition and introduces the dynamical core of RMOF. This layer is extended towards a set of concurrent state machines with several forms of interleaving and atomicity, operations, and observers in the extended variant of the core layer (see sections 2.3.3 - 2.3.5).

2.3.1 Core

The identified and introduced state machine subset corresponds to finite automata derived from UML2. Differences are, e.g., multiple actions on transitions in contrast to states and guards and effects based on the previously described action language (see section 2.2). Differences compared to UML2 state machines are, e.g., the definition of a formal action language and the omitting of state machine regions and events. The removal of regions and events was done to keep the language core to a minimum and postpone the definition/trigger the discussion of different variants in derived language layers. The behavior class and features and conclude by introducing the ingredients of a state machine defining behavior are introduced initially.

13.3.2 *Behavior* = (Class,{ (ownedAttribute, (specification, context, ownedParameter, context)), (isAbstract, *true*), (generalization, {Class})}) is a specification of how

its context classifier changes state over time. A classifier Behavior describes the sequence of state changes an instance of a classifier may undergo in the course of its lifetime. When a Behavior instance is associated as the method of a behavioral feature, it defines the implementation of that feature.

- ownedParameter = (Property,{ (aggregation, shared), (type, Parameter), (lowerValue, 0), (upperValue, *)}) references a list of parameters to the behavior that describes the order and type of arguments that can be given when the behavior is invoked and of the values that will be returned when the behavior completes its execution.

13.3.3 BehavioralFeature = (Class,{ (ownedAttribute, (ownedParameter)), (generalization, {Feature, Namespace})}) specifies that an instance of a classifier will respond to a designated request by invoking a behavior. BehavioralFeature is an abstract metaclass specializing Feature and Namespace. Kinds of behavioral aspects are modeled by sub-classes of BehavioralFeature.

- ownedParameter = (Property,{(subsettedProperty, {Namespace::ownedMember}), (type, Parameter), (upperValue, *)}) specifies the ordered set of formal parameters owned by this BehavioralFeature.
- method = (Property,{ (type, Behavior), (lowerValue, 0)}) is a behavioral description that implements the behavioral feature. There may be at most one behavior for a particular pairing of a classifier (as owner of the behavior) and a behavioral feature (as specification of the behavior).

There are three types of states available in RMOF, corresponding to start, normal, and end states.

15.3.2 FinalState = (Class,{ (generalization, {Vertex})}) A special kind of state signifying that the enclosing state machine is completed.

15.3.8 Pseudostate = (Class,{(ownedAttribute, (stateMachine)), (generalization, {Vertex})}) An abstraction that encompasses different types of transient vertexes in the state machine graph according to the UML2 specifications. In RMOF this state is only the entry state of a state machine.

- stateMachine = (Property,{(subsettedProperty, {Element::ownedMember}), (type, State Machine)}) references the state machine in which this Pseudostate is defined. This only applies to Pseudostates of the kind entryPoint or exitPoint.

15.3.11 State = (Class,{ (generalization, {Namespace, Vertex})}) in contrast to Pseudostate and Finalstate these define “normal” states in between.

A state machine holds the complete behavior consisting of states and transitions.

15.3.12 StateMachine = (Class,{(generalization, {Behavior})}) can be used to express the behavior of a part of a system.

- transition = (Property,{(aggregation, composite), (subsettedProperty, {ownedMember}), (type, Transition), (opposite, Transition.container), (lowerValue, 0), (upperValue, *)}) references the set of transitions owned by this state machine.
- subvertex = (Property,{(aggregation, composite), (subsettedProperty, {ownedMember}), (type, Vertex), (opposite, Vertex.container), (lowerValue, 0), (upperValue, *)}) references the set of vertexes that are owned by this state machine.

In contrast to UML2 where states can execute actions (e.g., as entry actions), RMOF supports only actions (and guards) on transitions. This has been done for two reasons. First, this approach is powerful enough and minimal according to finite automata. Second, some aspects of these UML inner state actions are not clear, e.g., atomicity.

15.3.14 Transition = (Class,{(ownedAttribute, (guard, effect, target, source, container)), (generalization, {NamedElement})}) A directed relationship between a source vertex and a target vertex.

- guard = (Property,{(aggregation, composite), (subsettedProperty, {ownedElement}), (type, String), (lowerValue, 0)}) is a constraint that provides a fine-grained control over the firing of the transition.
- effect = (Property,{(aggregation, composite), (subsettedProperty, {ownedElement}), (type, String), (lowerValue, 0), (upperValue, *)}) specifies an optional behavior to be performed when the transition fires.
- source = (Property,{ (type, Vertex), (opposite, Vertex.outgoing)}) designates the originating vertex (state or pseudostate) of the transition.
- target = (Property,{ (type, Vertex), (opposite, Vertex.incoming)}) designates the target vertex that is reached when the transition is taken.
- container = (Property,{ (aggregation, shared), (subsettedProperty, {namespace}), (type, State Machine), (opposite, State Machine.transition)}) designates the region that owns this transition.

15.3.16 *Vertex* = (Class,{(ownedAttribute, (outgoing, incoming, container)), (isAbstract, *true*), (generalization, {NamedElement})}) An abstraction of a node in a state machine graph. In general, it can be the source or destination of any number of transitions.

- outgoing = (Property,{(type, Transition), (opposite, Transition.source), (lowerValue, 0), (upperValue, *)}) specifies the transitions departing from this vertex.
- incoming = (Property,{(type, Transition), (opposite, Transition.target), (lowerValue, 0), (upperValue, *)}) specifies the transitions entering this vertex.
- container = (Property,{ (aggregation, shared), (subsettedProperty, {namespace}), (type, State Machine),}) references the region that contains this vertex.

2.3.2 Formal Semantics

The RMOF behavior relies on the definition of a so called "Symbolic Transition System" (STS) which is defined (including its runs) in the following two definitions.

Definition 24 (STS) *A symbolic transition system (STS) $S = (V, \Theta, \rho)$ consists of V, a finite set of typed system variables, Θ, a first-order predicate over variables in V characterizing the initial states, and ρ, a transition predicate, that is a first-order predicate over V, V', referring to both primed and unprimed versions of the system variables (their current and next states).* ■

The STS induces a transition system on the set of interpretations of its variables as follows.

Definition 25 (Runs of an STS) *Let $S = (V, \Theta, \rho)$ be an STS and $\mathcal{T}$ the set of types of variables in V. Let $\mathcal{D}_\tau$ be a semantic domain for each $\tau \in \mathcal{T}$.*

(i) A snapshot

$$s : V \to \bigcup_{\tau \in \mathcal{T}} \mathcal{D}_\tau$$

of S is a type-consistent interpretation of V, assigning to each variable $v \in V$ a value $s(v)$ over its domain. Σ denotes the set of snapshots of S.

(ii) A snapshot $s \in \Sigma$ inductively defines the value $[\![expr]\!](s)$ for first-order predicates 'expr' over V and the value $[\![expr]\!](s, s')$ for first-order predicates 'expr' over V, V', where s provides the interpretation of unprimed and s' the interpretation of primed variables in 'expr'.

(iii) A snapshot $s \in \Sigma$ is called initial, iff $[\![\Theta]\!](s) = true$.

(iv) Let $s, s' \in \Sigma$ be snapshots of S. Snapshot s' is called S-successor of s, iff $[\![\rho]\!](s, s') = true$.

(v) *A computation, or run, of S is an infinite sequence of snapshots $r = s_0\, s_1\, s_2 \ldots$, satisfying the following requirements:*

- *Initiation:* *s_0 is initial.*
- *Consecution:* *Snapshot s_{j+1} is an S-successor of s_j, for each $j \in \mathbb{N}_0$.*

(vi) *The set of all computations of S is denoted as $runs(S)$. $r(i)$ is used to denote the i-th snapshot of a run $r \in runs(S)$ and*

$$r/i := r(i)\, r(i+1)\, r(i+2) \ldots$$

to denote the infinite suffix starting at $r(i)$, $i \in \mathbb{N}_0$. ■

The set of all transitions Tr_C of all classes $c \in C$ is defined as follows: $Tr_C \stackrel{df}{=} \bigcup_{c \in C} c.Tr$.

Analogous is the set of all objects $O_C \stackrel{df}{=} \bigcup_{c \in C} O_c$.

The boolean flag *sysfail*is introduced to indicate an undefined state of the system. Normally this flag is set to *false* and if e.g., an expression is evaluated to $\perp$ then *sysfail* is set to *true*[1].

Definition 26 (System Semantics) *Let $\overset{\circlearrowleft}{M}$ be a system. The semantics of $\overset{\circlearrowleft}{M}$ is defined as:*

$$STS(\overset{\circlearrowleft}{M}) = (V, \Theta, \rho),\ where$$

System variables: $V := \{sconf : \mathcal{T}_{sconf}(\overset{\circlearrowleft}{M}), \overset{\circlearrowleft}{m}.c : \} Q, sysfail : \mathbb{B}\}$.

Initial condition: $\Theta := \overset{\circlearrowleft}{m}.c = q_0 \wedge sysfail = false$

Transition relation: *The intermediate predicate ρ_0 composes the above introduced sub-predicates and additional conditions on their application within object's life-cycle as follows:*

$$\begin{aligned}\rho_0 :=& (\neg sysfail \wedge \exists (q, \gamma, q') \in \overset{\circlearrowleft}{m}.tr : \overset{\circlearrowleft}{m}.c = q \wedge \overset{\circlearrowleft}{m}.c' := q' \\ & \wedge (assign(\gamma) \vee guard(\gamma)) \wedge q \neq q_x) \vee q = q_x\end{aligned}$$

The final transition relation ρ is obtained from ρ_0 by adding a frame axiom which requires that only those places of s are allowed to change in the transition to s', which get new values by an assignment ":=" in ρ_0, and changing the assignments to "=". The semantics of a System S is given as the set $runs(STS(S))$ of all computations in S (starting at Θ). ■

A single state machine can be executed with this semantics. The next section extends these definitions and specifies the extended dynamical core of RMOF.

[1] And remains *true* for the rest of the execution.

2.3.3 Extended Core

The extended dynamical core inherits the RMOF core and adds the so called "global state machine". The global state machine supports, e.g., different interleavings and atomicities, operations, and observers.

The global state machine handles primarily the determination of the execution context to identify the next effect to be executed. The determination is based on the interleaving of all active state machines, the atomicity of the current state machine, it's current state and enabled transitions. The enabled state of a transition is based on the current valuations of relevant attributes. If multiple transitions are enabled one is non-deterministically chosen.

The extended dynamical core introduces operations. An operation consists of a set of attributes/parameters to exchange values with the calling state machine, a state machine defining the behavior of the operation itself, and a state machine handler managing the execution of the state machine. All operation state machines are initially suspended. When called, the parameters of the operation are assigned, the calling state machine is suspended and the called state machine of the operation executed (more precise: it is included in the set of all active state machines by setting it from "suspended" to "active"). After the state machine of an operation reached the final state, the "inout" parameters and the "return" parameter are set, the called state machine is reset, suspended and calling state machine is continued. In the following text two additional classes covering behavior in general are introduced.

13.3.2 *Behavior-Extended* = (Class,{ (ownedAttribute, (specification, context, ownedParameter, context)), (isAbstract, *true*), (generalization, {
}) Class}) is a specification of how its context classifier changes state over time. A classifier behavior describes the sequence of state changes an instance of a classifier may undergo in the course of its lifetime. When a behavior is associated as the method of a behavioral feature, it defines the implementation of that feature.

- ownedParameter = (Property,{ (aggregation, shared), (type, Parameter), (lowerValue, 0), (upperValue, *)}) references a list of parameters to the behavior that describes the order and type of arguments that can be given when the behavior is invoked and of the values that will be returned when the behavior completes its execution.

13.3.3 BehavioralFeature-Extended = (Class,{ (ownedAttribute, (ownedParameter)), (generalization, {Feature, Namespace})}) specifies that an instance of a classifier will respond to a designated request by invoking a behavior. BehavioralFeature is an abstract meta-class specializing Feature and Namespace. Kinds of behavioral aspects are modeled by sub-classes of BehavioralFeature.

- ownedParameter = (Property,{(subsettedProperty, {Namespace::ownedMember}), (type, Parameter), (upperValue, *)}) specifies the ordered set of formal parameters owned by this BehavioralFeature.
- method = (Property,{ (type, Behavior), (lowerValue, 0)}) defines a behavioral description that implements the behavioral feature. There may be at most one behavior for a particular pairing of a classifier (as owner of the behavior) and a behavioral feature (as specification of the behavior).

The next subsection defines operations as a specialization of these general behavior elements.

Operations and Constraints

All operations can use parameters to communicate with the calling context. All parameters are defined with ParameterDirectionKind. The ParameterDirectionKind specifies if the parameter is read, written, or both when the operation is invoked and when the state machine of the operation finished the execution and returns the control to the state machine that invoked the operation.

7.3.36 Operation = (Class,{(ownedAttribute, (bodyCondition, ownedParameter, postCondition, preCondition, stateMachine, calledFromStateMachine)), (generalization, {Behavioralfeature})}) is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

- ownedParameter = (Property,{ (aggregation, composite), (type, Parameter), (upperValue, *)}) specifies the parameters owned by this operation.
- postCondition = (Property,{ (aggregation, composite), (type, Constraint), (subsettedProperty, {Namespace::ownedRule}), (lowerValue, 0) (upperValue, *)}) specifies an optional set of constraints specifying the state of the system when the operation is completed.
- preCondition = (Property,{ (aggregation, composite), (type, Constraint), (subsettedProperty, {Namespace::ownedRule}), (lowerValue, 0) (upperValue, *)}) specifies an optional set of constraints specifying the state of the system when the operation is invoked.
- stateMachine = (Property,{ (aggregation, composite), (type, State Machine), (lowerValue, 0), (upperValue, 1)}) specifies the state maschine of the operation.
- calledFromStateMachine = (Property,{(type, State Machine), (lowerValue, 0), (upperValue, 1)}) stores the state maschine that called the operation.

7.3.41 Parameter = (Class,{ (ownedAttribute, (default, direction, operation, defaultValue)), (generalization, {MultiplicityElement, TypedElement})}) is a specification of an argument used to pass information into or out of an invocation of a behavioral feature.

- direction = (Property,{(defaultValue, in), (type, ParameterDirectionkind)}) indicates whether a parameter is being sent into or out of a behavioral element.
- operation = (Property,{ (aggregation, shared), (subsettedProperty, {NamedElement::namespace}), (type, Operation)}) references the operation owning this parameter.
- defaultValue = (Property,{(subsettedProperty, {Element::ownedElement}), (type, String), (lowerValue, 0)}) specifies a string that represents a value (computation description) to be used when no argument is supplied for the parameter.

7.3.42 ParameterDirectionKind = (Enumeration,{ (ownedLiteral, ('in', 'inout', 'out', 'return'))}) Parameter direction kind is an enumeration type defining the following literals used to specify direction of parameters:

in indicates that parameter values are passed into the behavioral element by the caller.

inout indicates that parameter values are passed into a behavioral element by the caller and then back out to the caller from the behavioral element.

out indicates that parameter values are passed from a behavioral element out to the caller.

return indicates that parameter values are passed as return values from a behavioral element back to the caller. In contrast to out parameters the operation call is substituted with the return value.

Constraints are introduced in this section because they are specified as operation with a fix return type mapped on $\mathbb{B}$. Constraints constraint elements by evaluating boolean expressions assigned to an element.

7.3.10 Constraint = (Class,{ (ownedAttribute, (constrainedElement, context, specification)), (generalization, {PackageableElement})}) is a condition (a boolean expression) that restricts the extension of the associated element beyond what is imposed by the other language constructs applied to the element.

- constrainedElement = (Property,{(type, Element), (upperValue, *)}) specifies a set of Elements referenced by this constraint.

- context = (Property,{ (isDerivedUnion, *true*), (type, Namespace), (lowerValue, 0)}) specifies the namespace that is the context for evaluating this constraint.
- specification = (Property,{ (aggregation, composite), (subsettedProperty, {Element::ownedElement}), (type, Operation)}) must be true when evaluated in order for the constraint to be satisfied.

The global state machine supports a set of concurrently executed state machines. Each can have its own level of atomicity defined by the so called "run-to-completion" (RTC) level of the state machine. RTC="statemachine" fires transitions as long as possible. RTC="transition" stops after a single transition was fired. After each stop the next execution context is evaluated. RTC="effect" stops after a single sub effect was fired. No executable effects (meaning all relevant transition guards are evaluated to "false") results in an execution error. A "global step" is done if all active state machines have been triggered by the global state machine.

All state machines are managed in so called "state machine handlers". A state machine handler is an object of the class shown in figure 2.5. State machine handlers

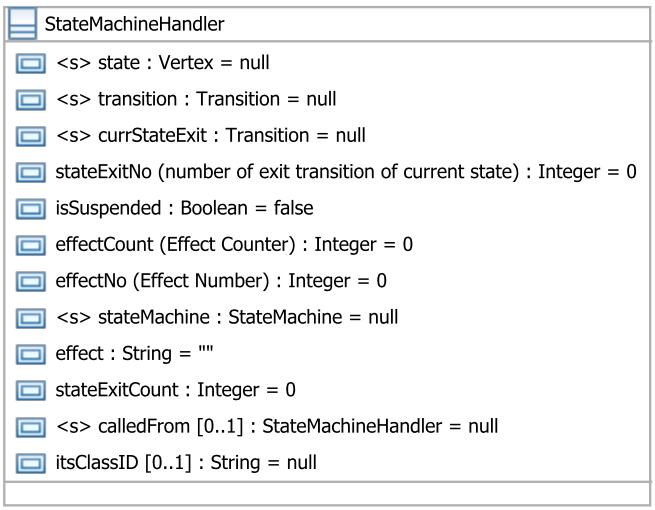


Figure 2.5: State machine handler

store, e.g., if the state machine is executed and if it is executed the current state, transition effect etc.

A special variant of a state machine is an observer. Observers are used to check behavior (described in detail in section 2.4) of other state machines instead of speci-

fying behavior itself. An observer has an RTC="observer" firing transitions as long as the final state is not reached. If no transition can be fired and the final state is not reached the observer is suspended. All defined observers are executed initially before all state machines switched and after all other state machines switched (according to their atomicity). In order to handle concurrent state machines the global state machine instantiates state machine handlers handling all execution related / dynamic variables of a state machine like the current transition or current set of transitions.

The initial state is represented by a black filled circle. The final state is represented by a black filled circle surrounded by a white circle with a black border line. Each transition can have a transition attachment consisting of a guard written in squared brackets and a set of effects prefixed by a gearwheel and suffixed/separated by a semicolon. A set of effects prefixed by a gearwheel is the minimal atomic unit on all RTC levels.

The global state machine starts by instantiating a state machine handler for all initially active state machines. A state machine handler covers all dynamic aspects of a state machine, e.g., to store the active state, transition, and effect. This enables the concurrent execution of the same state machine by multiple state machine handlers. The execution of the global state machine stops if no state machine handler is active. Figure 2.6 shows the initialization phase of the global state machine. All observers are

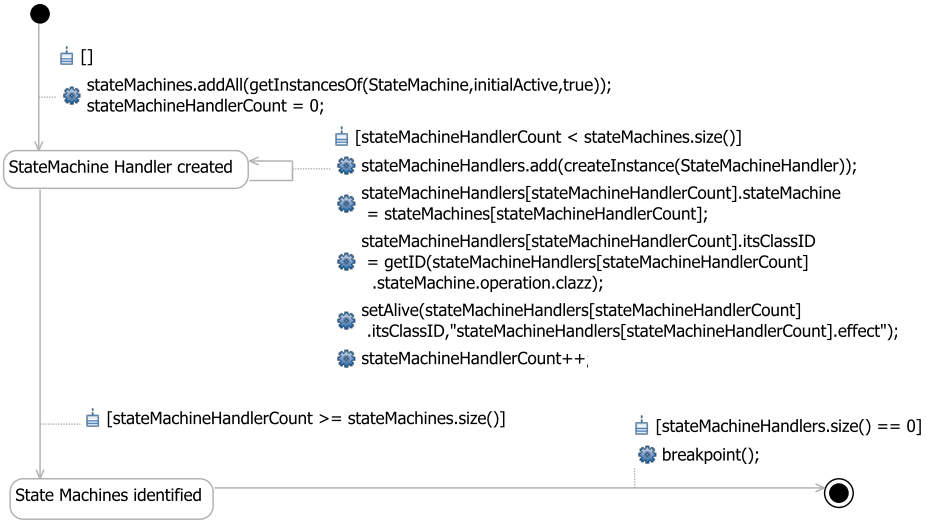


Figure 2.6: Initialization of the global state machine

executed before the state machines are executed in all execution steps. An observer runs as long as an effect is executable. If no effect is executable, an observer is sus-

Figure 2.7: Observer check of the global state machine

pended until the next global step is executed. Figure 2.7 shows the observer checking if only observers are left in the execution pipeline. If this is the case the execution stops because observers don't change attribute values. The checking takes place after the observers have been executed. After the current set of active state machines has been identified all observers are executed followed by all active state machines. All state machines are executed by determining the current state first. If the current state of the state handler is null, the current state is the initial state otherwise the state handler points to the current state from the last step. If the current state is the final state, the execution is done and the state machine is suspended. If the state machine was part of an operation, the operation call part in the calling state machine is substituted by the return value and the assigned attributes of the out and inout parameters are set. If the current state is not the final state, the enabled (outgoing) transitions are determined and the first one that is evaluated to true (non-deterministically chosen) is set to be the current transition. If there are sub effects from a previous execution that need to be processed, the transition determination is skipped. If a new transition has been determined, the first sub effect is identified and the transition is colored (in a simulation run). Figure 2.8 shows the determination of the next sub effect to be executed. Figure 2.9 shows the effect handling. After the sub effect is identified it is (colored in the graphical environment and) executed. If there exist more sub effects

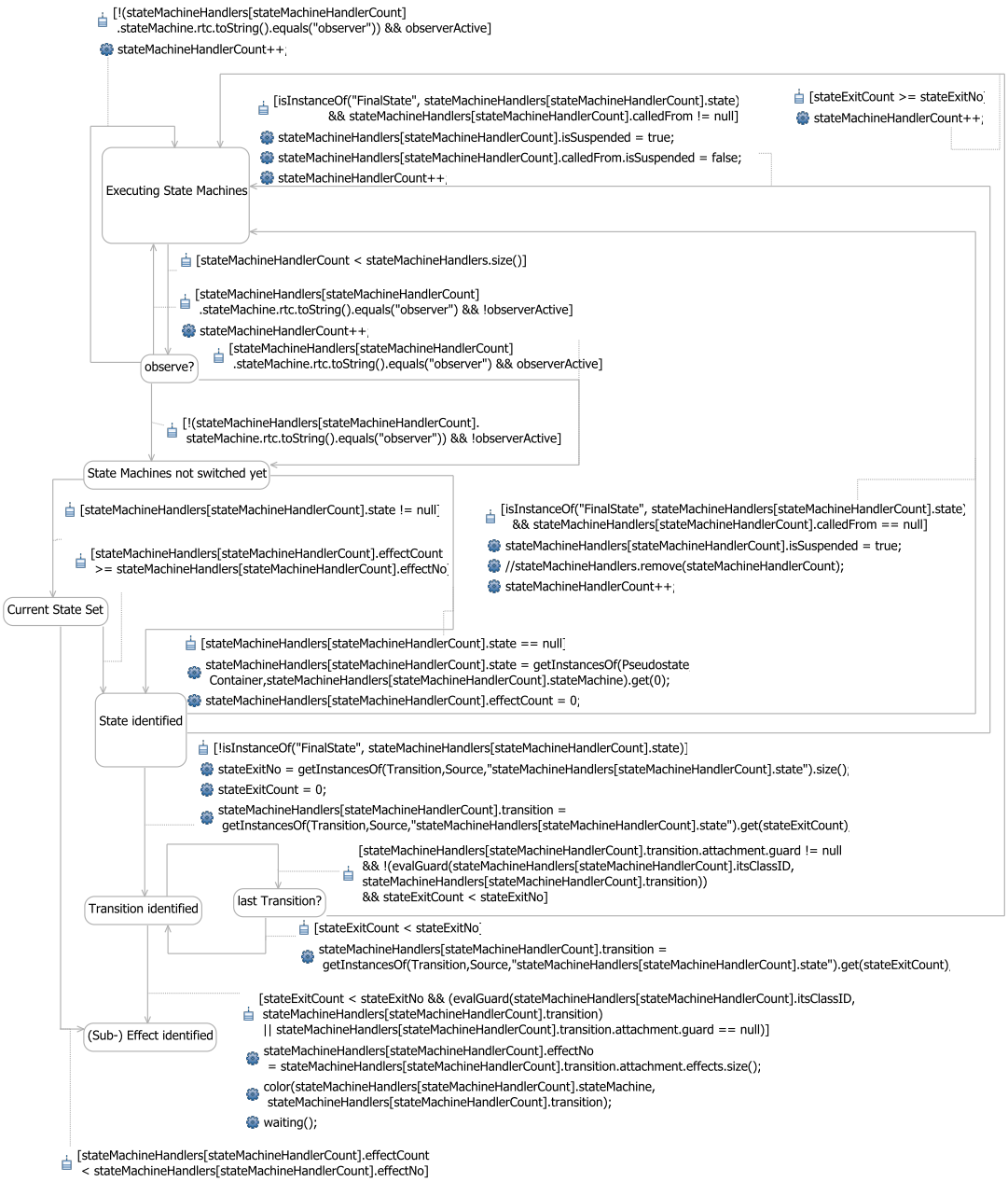


Figure 2.8: Sub effect identification of the global state machine

on the current transition it depends on the atomicity of the state machine, if the next sub effect is executed or not. If all sub effects have been executed (the current effect

Figure 2.9: Global state machine executions

counter is larger than the number of the available effects), the current state is set on to the transition target (and colored) and the effect counters are reset. A state machine changes its state from active to inactive when it has reached its final state. When no active state machines are available the execution of the global state machine stops.

The global state machine supports breakpoints, e.g., to investigate information during a debugging session, it handles the coloring of graphical elements, and is able to wait a predefined time (for demonstration purposes). The following section defines the computation of some predefined operations in RMOF, e.g., to compute derived values

or derived union sets. In contrast to the primitive operations that have been directly defined mathematically, complex operations are represented as state machines.

2.3.4 Derived Values

Derived attributes are derived/computed from other attributes. The OMG specifications do not define how to compute them.

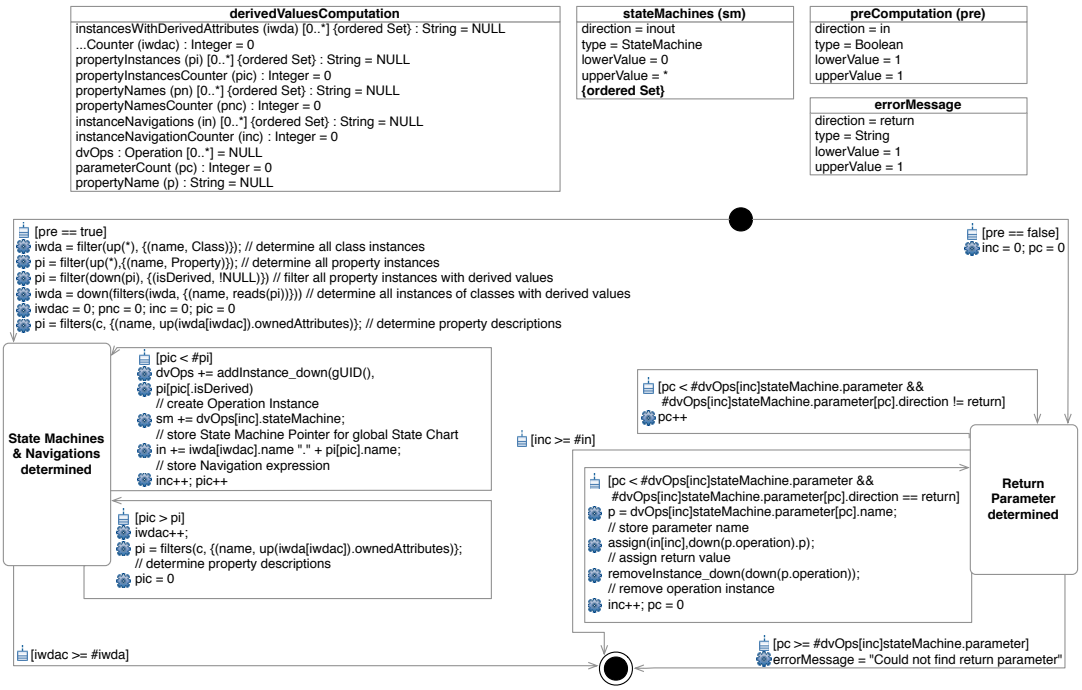


Figure 2.10: Derived values computation

In RMOF the value of a derived attribute is the result of the execution of an assigned operation and computed after a global step has taken place and before observers are executed. Figure 2.10 defines the state machine to compute derived values. The specified attributes and their classes are depicted in the top of the figure. The state machine starts by identifying all derived value attributes. The corresponding state machines of the assigned operations of the derived values are invoked and their return value is assigned to the derived value attribute. The computation order of derived values is non-deterministically chosen in the current semantics. This could be changed towards a partial ordering defined by computation dependencies.

An example of a derived value is the qualified name of an attribute. Qualified names

are prefixed with all namespace names that contain (transitively) the NamedElement. The state machine is presented in figure 2.11. The qualified name of a NamedElement

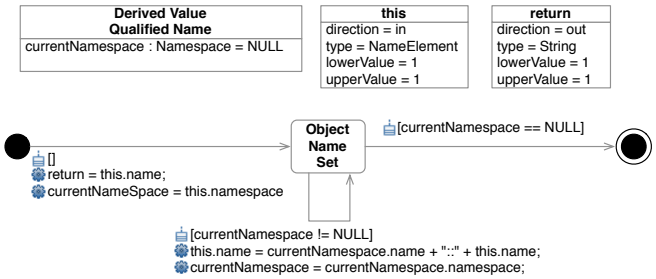


Figure 2.11: Derived value - qualified name

is build successively by concatenating all namespace names, separated by "::" and appendixed by the element name itself. Another predefined variant of a derived value

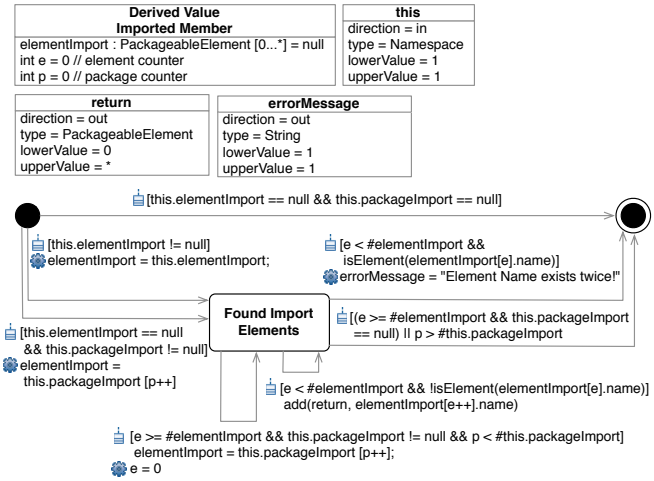


Figure 2.12: Derived value - ImportedMember

are member imports. PackageableElements can be imported into another package by so called package/member imports. Whereby the imported element is a reference to the source element, not a copy of the element. Figure 2.12 shows the corresponding state machine importing namespace elements. Elements are neglected during an import if already an element with such a name exists in the importing namespace. In contrast to derived values derived unions form a kind of superset. Based on [63] the semantics of derived unions is defined as superset of all sub-setting variables. Figure 2.13 shows the corresponding state machine. The domain of a derived union attribute is defined as subset of the domains of all sub-setting attributes. The current value of a derived

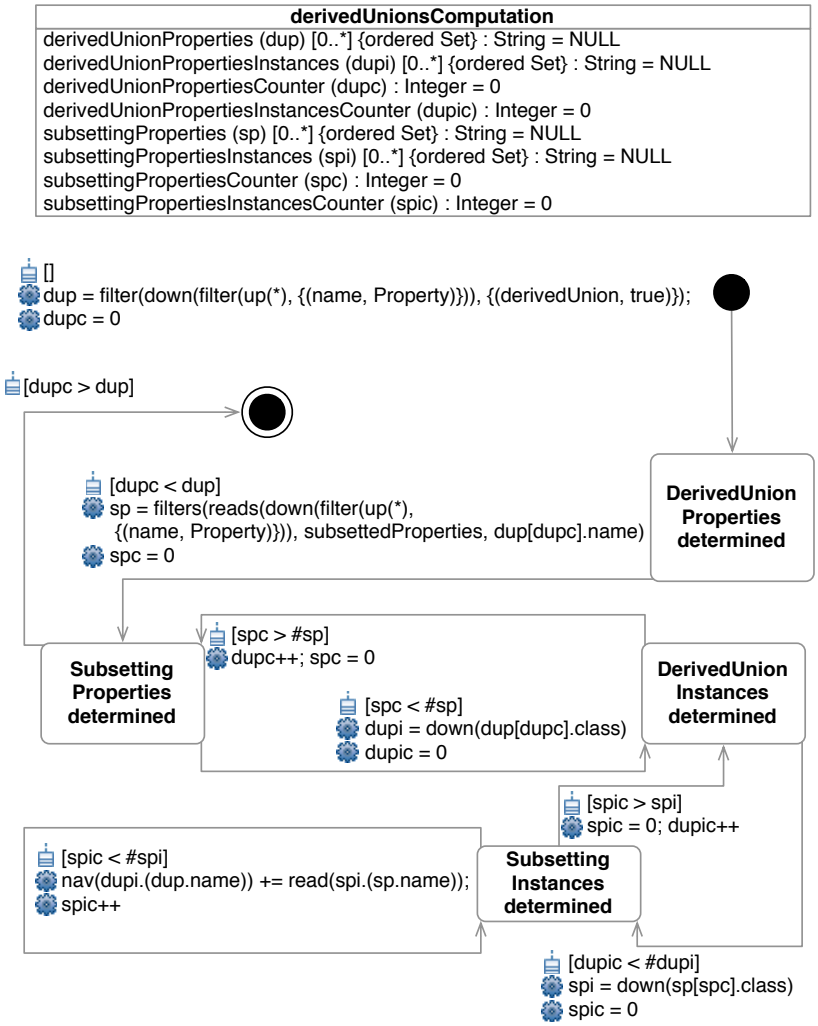


Figure 2.13: Derived unions

union attribute equals the current values of all sub-setting attributes. The next section introduces the predefined state machines handling operation calls.

2.3.5 Operations

The invocation of an operation is done in several steps. The first step determines the so called “inner most” complex operation. During an invocation of a (possibly) nested operation call the “inner most” complex operation is determined, executed, and substituted by the result of the operation. The determination of the inner most complex

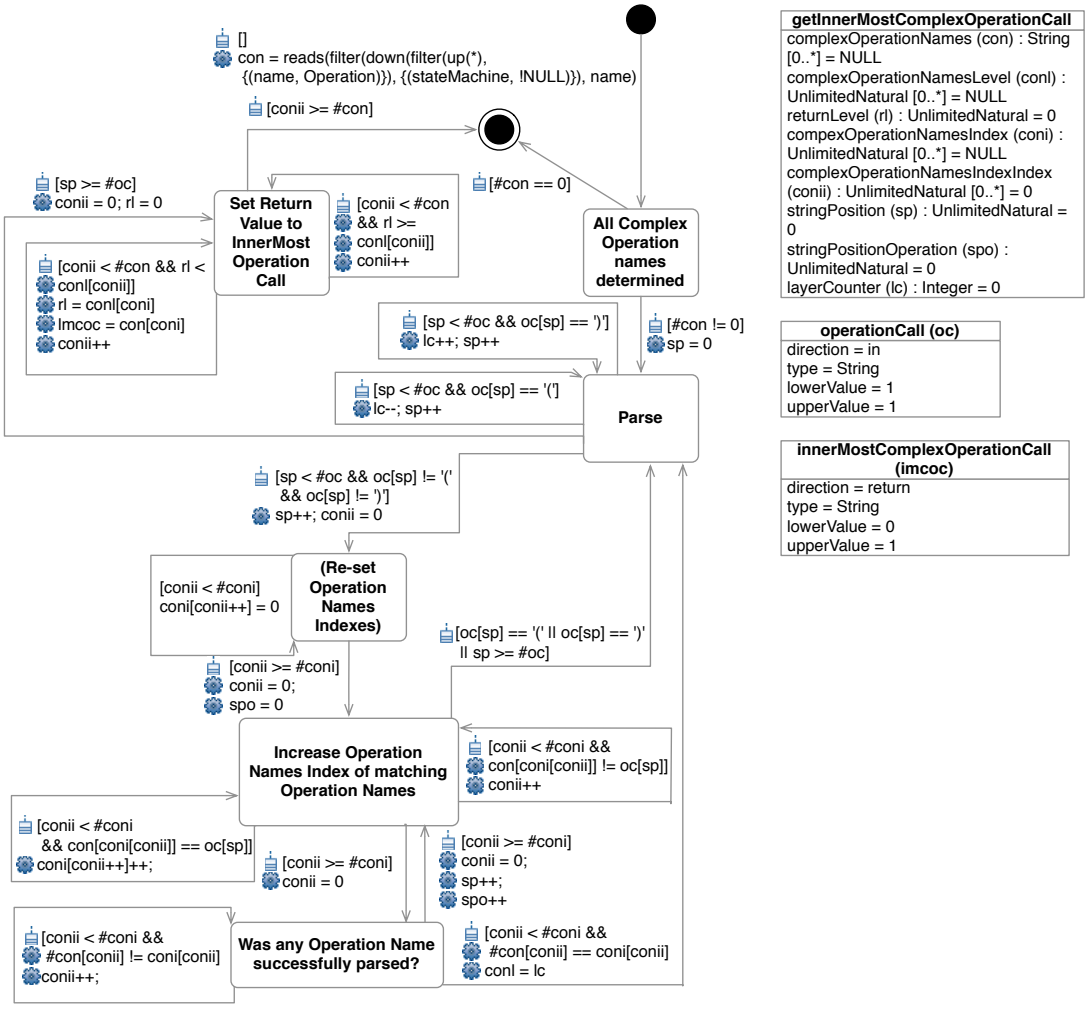


Figure 2.14: GetInnerMostOperationCall

operation is presented in figure 2.14. The inner most complex operation is the complex operation that can be executed first of a nested operation call because it does not call another complex operation. By successively evaluating the inner most complex operations and substituting the operations calls with their return values an executable evaluation is guaranteed. In order to find the inner most complex operation the operation call string is disassembled into (valid) operation names and parameters. The inner most complex operation is the one with the first closing brackets.

After the inner most complex operation has been identified, the operation parameters

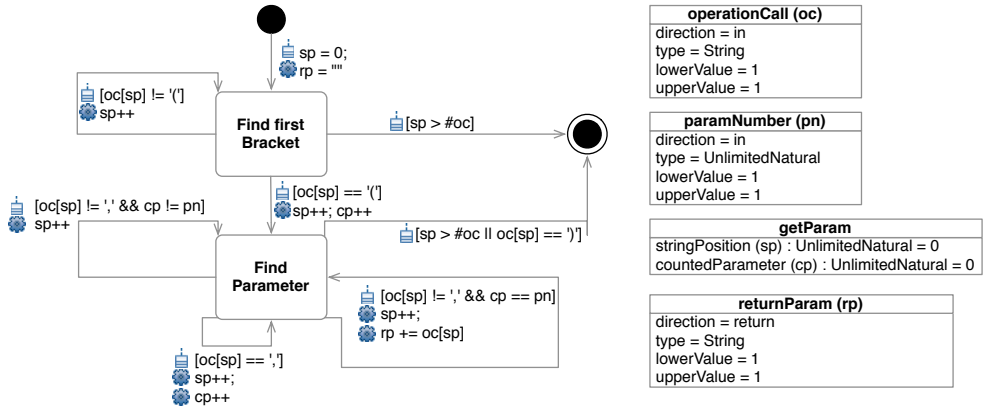


Figure 2.15: GetParam

are parsed and assigned to the operation parameters via “getParam” and “setParam” operations. Figure 2.15 presents the state machine to get parameters. Getting parameters is done during the invocation of an operation to assign the local attributes of the operation that can be used by the state machine. Figure 2.16 presents the state machine to set parameters. The next section addresses aspects of temporal logics that

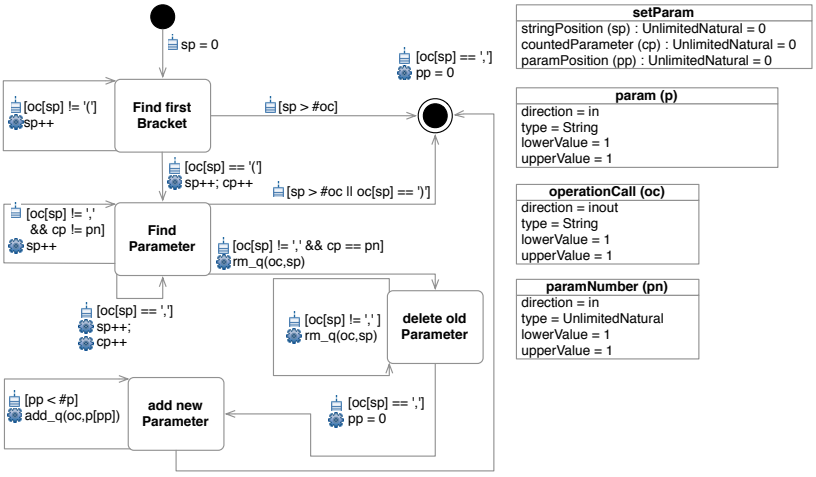


Figure 2.16: SetParam

are used to specify a special type of state machine called observer.

2.4 Observers

The RMOF extended dynamical core supports discrete time in terms of a (global) step counter. But each instance layer can define own timing models, in particular ones that base on continuous time. In this sections some observers defining discrete Linear Temporal Logic operators are presented. In the examples a variable t is used to represent the current time and $t+i$ a time in the future. The variable r represents the return value of the observer evaluation. The following temporal operations introduced:

- neXt φ : φ has to hold at the next state/interval. Figure 2.17 represents the corresponding observer state machine.

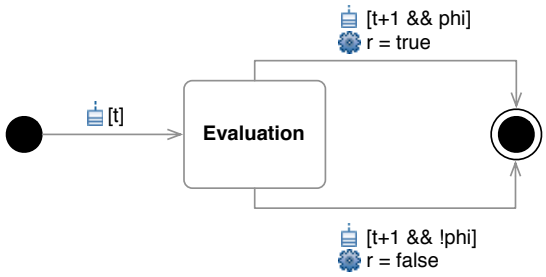


Figure 2.17: Temporal observer realizing "next"

- Globally φ : φ has to hold on the entire subsequent execution path. Figure 2.18 represents the corresponding observer state machine.

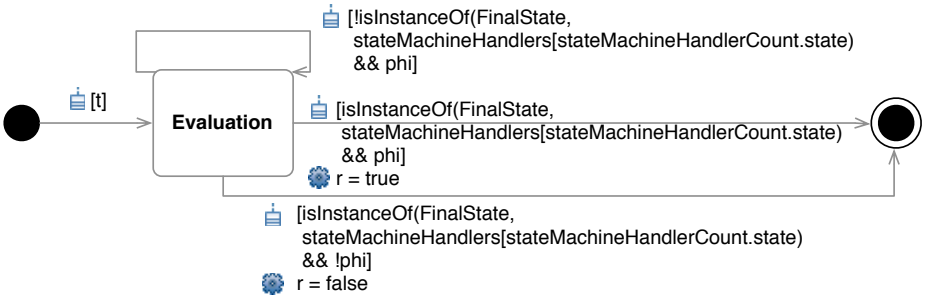


Figure 2.18: Temporal observer realizing "globally"

- Eventually in the future φ : φ has to hold somewhere on the subsequent execution path. Figure 2.19 represents the corresponding observer state machine.

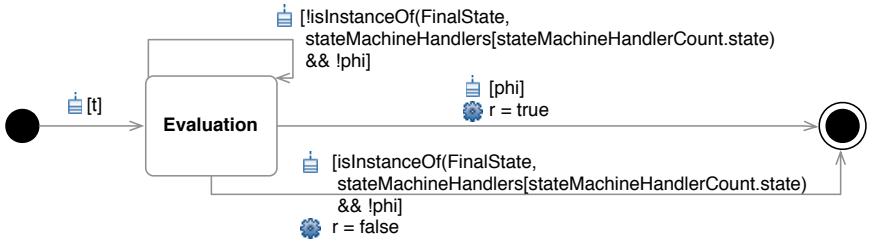


Figure 2.19: Temporal observer realizing "eventually in the future"

- υ Until φ: υ has to hold at least until φ, which holds at the current or a future position on the subsequent execution path. Figure 2.20 represents the corresponding observer state machine.

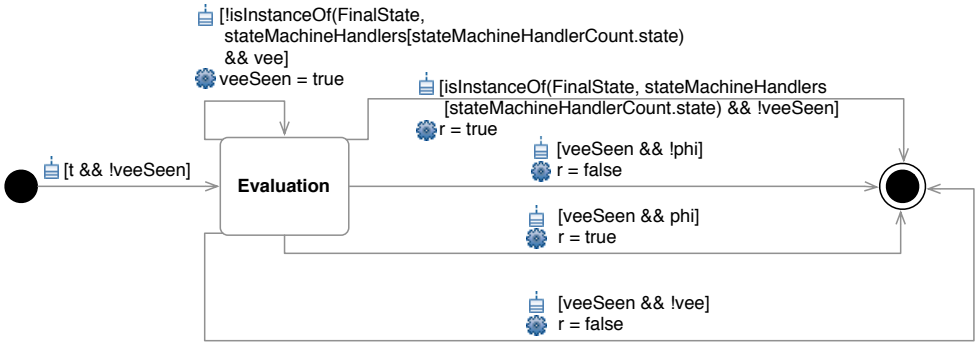


Figure 2.20: Temporal Observer realizing "until"

- υ Release φ: At the first position in which υ is true, φ ceases to be true and is to be required to be true until release occurs. Figure 2.21 represents the corresponding observer state machine.

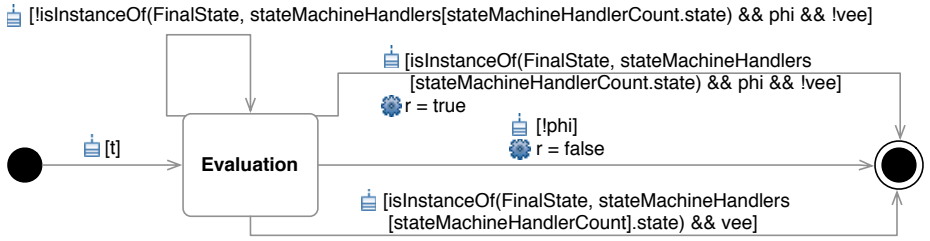


Figure 2.21: Temporal observer realizing "release"

The next section shows an implementation of this language frame allowing for example the graphical specification of modeling layers, debugging, and platform binding to explore all kinds of models and languages to form a process modeling framework with the required flexibility to realize suitability and sustainability of derived process models.

2.5 Implementation

There are several challenges that must be faced when implementing RMOF. The Eclipse platform [1] has been identified to meet these challenges. The Eclipse platform offers a flexible plugin concept based on the Open Service Gateway initiative (OSGi [2]) specification, which is intensely used in the prospering Eclipse community offering more or less all ingredients to implement RMOF. The following challenges/requirements are addressed in the implementation:

- Domain model creation for arbitrary models is realized on base of the Eclipse Modeling Framework (EMF [64, 65]). EMF supports the creation of domain models on base of a syntax comparable to the Meta Object Facility of the OMG. Models can be created from scratch or by using import functions that are able to parse Rational Rose models [3], XML schema, and annotated Java. EMF and additional plugins offer:
  - XMI [66] support, which is important to support in particular all OMG specification (like UML2 [54, 55]), is directly offered by the EMF Framework.
  - Models to text/code generation by so called Java Emitting Templates (JET [67]). JET supports JSP-like template files that can be edited and transformed into any kind of source artifact including Java, HTML, properties or XML files. There exist additional "model2text" frameworks like Acceleo [68] which is an implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL [69]) standard or Xpand [70], a statically-typed template language. The first and the last project even offer particular editor environments to support the editing of the generator files.
  - The retrieval of model information comparable to SQL (Structured Query Language) statements with the Model Query Project [71].
  - Model compare and merge can be done with the Model Compare Project [72]. In the first step called "matching" similar model parts are identified, in the second step called "differencing" the delta between the models is identified.

[1] www.eclipse.org
[2] www.osgi.org
[3] http://www-01.ibm.com/software/rational

  - The definition of constraints is possible in form of Java and OCL, model validation, and validation support of event listeners is offered by the Model Validation Project [73].

- The creation of user-defined graphical editors on base of EMF Models is supported by the Graphical Modeling Framework (GMF [74, 75]). With GMF it is possible to define geometric objects and images, Tool palettes and map them to defined EMF models to generate editor source code in form of an Eclipse plugin. There exist other projects building GMF based editors for graphical oriented languages like UML2 [76], BPMN [77] (Business Process Model and Notation), or the IMM [78] (Information Management Meta-model).

- Model transformation is supported by ATL (Atlas Transformation Language [79]) including editor and debugging facilities. The projects Declarative QVT [80] and Operational QVT [81] support Model Transformation on base of the Query/View/-Transformation Specification [82] of the OMG. Operational QVT supports a kind of imperative language to specify explicit steps to execute on order to describe transformations in contrast to the Declarative QVT that assumes a direct correspondence between source and target model parts.

- Model versioning is supported by the Teneo Project [83]. Teneo is a database persistency solution for EMF. It supports automatic creation of EMF to relational mappings.

- IDE support of all editor instances is fulfilled because all generated editor sources (of EMF and GMF as main source) are Eclipse plugins itself. Thus other Eclipse plugins e.g., to support version control, can also be used in these editors.

- Distribution and collection of process information is realizable via Net4J [84] and CDO [85]. Net4J is an extensible client-server system based on the Eclipse and the Spring framework. It is possible to extend the protocol stack with Eclipse plugins that provide new transport or application protocols. CDO is both a technology for distributed shared EMF models and a fast server-based O/R mapping solution. With CDO it is possible to enhance existing models in such a way that saving a resource transparently commits the applied changes to a relational database with Net4J over the network. Optionally other connected clients are actively notified about these changes so that their model copies get partially invalidated and all user interfaces reflect the current state at once.

- (Semi-) Automatic retrieval and postulation/enactment of process information in connected development environments. Due to the wide range of development activities covered by Eclipse and its plugins it is possible to connect a single

environment to retrieve and postulate process information directly for all these process steps.

RMOF is realized on base of Eclipse including mainly EMF, GMF, and CDO. Figure 2.22 shows the approach on an abstract level. Each meta-modeling layer in RMOF is mapped on an Ecore meta-model which is an instance of Ecore. Such a mapping can be done in two ways, statically and dynamically. During a static model creation the Ecore is created in a manual domain model creation process in the Eclipse framework and code generators synthesize the editor and simulation source code (e.g., factories, notification mechanisms, and XML based persistence mechanisms). All GMF based elements require a static mapping because their representation must be (pre-)defined. A dynamic mapping is supported by designated EMF APIs to create a meta-model from source code.

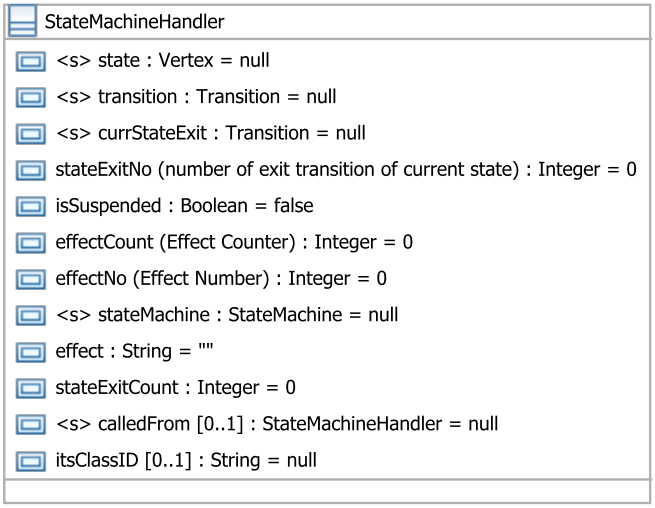


Figure 2.22: RMOF concept vs. implementation

RMOF uses a hybrid approach. All GMF mapped elements must be statically mapped, the rest can be dynamically instantiated and the required source code is generated and merged with the static source code during run time. This requires some implementation twists because all objects should be able to "see" each other regardless if based on a statically mapped part or a dynamically mapped part of the meta-model. All effects are mapped (currently) onto Java code, compiled together with the EMF classes and executed directly by a Java Virtual Machine to ensure maximal performance in the editor environment. The simulation environment of RMOF supports a composition of different meta-model layers, their execution/simulation and in particular their debugging. Figure 2.23 shows a screenshot during a debug run of the RMOF

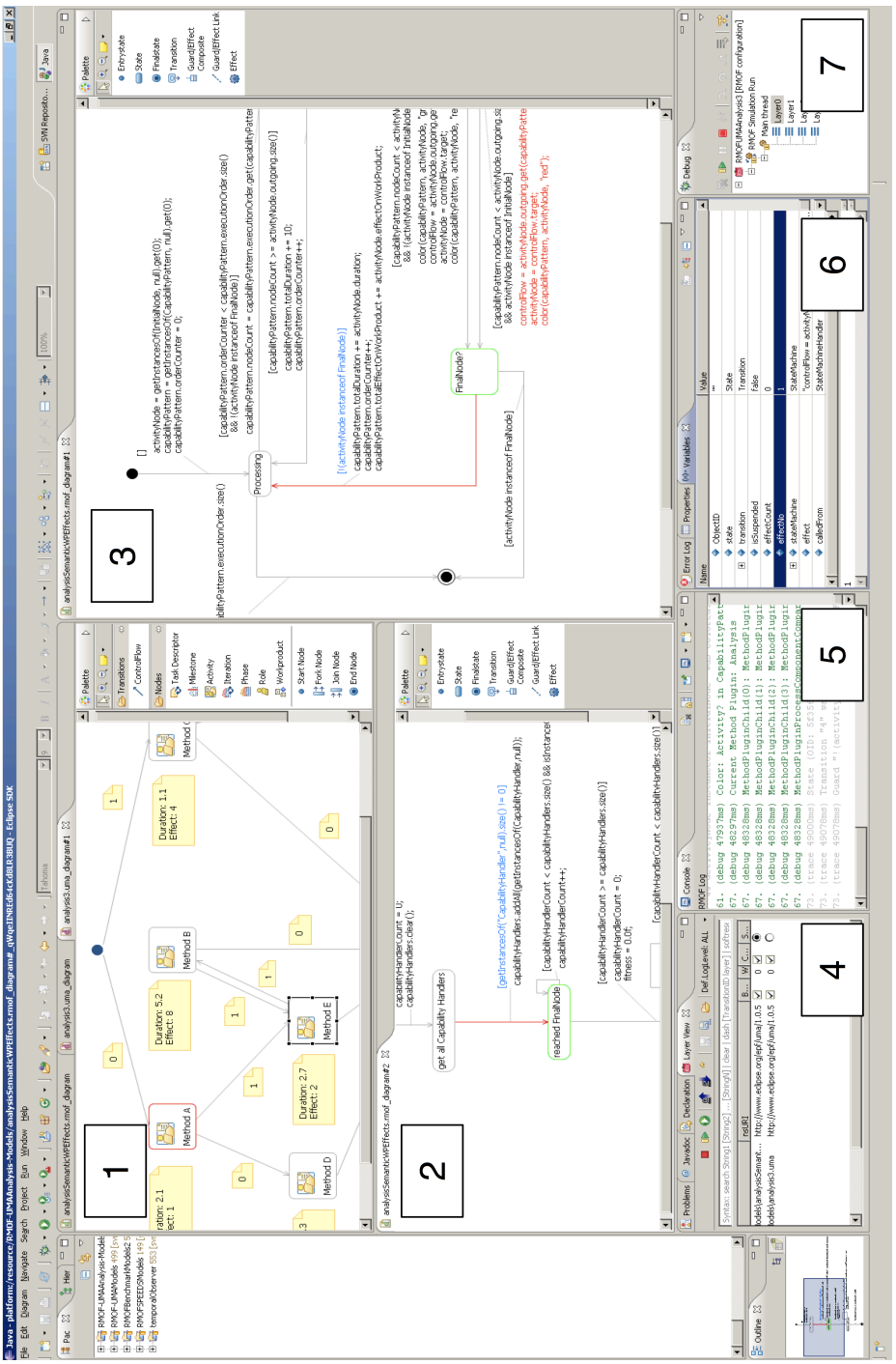


Figure 2.23: RMOF environment screenshot

environment. Sub-Figure 1 shows a SPEM Editor executing different methods in a process model. Sub-Figure 2 shows an observer checking if the final node of the SPEM model is already reached. Sub-figure 3 shows an RMOF model editor in which the SPEM semantic has been described in terms of an RMOF State Machine. Sub-figure 4 shows the layer view that can be used to compose language layers. Additionally this layer is able, e.g., to execute step-wise for all modeling layers, color a model layer, and wait (a certain amount of milliseconds on a wait point). Sub-figure 5 shows the console view of RMOF that can postulate all kinds of log information - if required separated for each language layer. Sub-figure 6 shows the variable view of the debug views of RMOF that can be used to retrieve all variable valuations during run time. Sub-figure 7 shows the debug view that can also be used to restrict the variables shown in the variable view regarding the layer that is chosen in the debug view.

The RMOF environment is able to access all EMF based editors, e.g., to highlight the current state of a state chart during a simulation. All constructs supported by RMOF based on Java are compiled before a simulation starts and invoked by using Java reflections to ensure maximal performance. During the compilation additional actions ensure that, e.g., the variables of the different language layers are defined and getter/setter of the variable are used during an assignment in the action language. The RMOF environment allows a distribution of the models over the network to distribute (and collect) live process information.

The implementation of RMOF is done completely in Java for the simulation/editor environment, but can be easily changed towards other platforms, e.g., to support additional data types or operations with a higher precision or other analysis methods (e.g. model checking). There are powerful code generation facilities for this purposes available.

The Software Process Engineering Metamodel (SPEM) will be instantiated as RMOF instance and complemented in the next chapter and will constitude a process frame to embed the synthesize process interactions/models introduced in the following chapters. Figure 2.24 shows an example RMOF instance and a run of the symbolic transition system describing the semantics of the instances. Figure 2.24 shows from left to right the data structures in terms of a class diagram, the behavior in terms of an RMOF State Machine, and the semantics in terms of the corresponding run of the underlying symbolic transition system. The class diagram specifies a “Test Class” with an attribute i of type Integer and and initial value of 0 and a “Test Operation” with a “Test Chart”. The “Test Chart” is represented in the “State Machine”. The “State Machine” increases i initially and continues while i is below 5. The “Run of an STS” shows all variables including i, a variable end - indicating if the final transition was reached and a variable $fail$ - indicating a failure state (e.g., a division by 0).

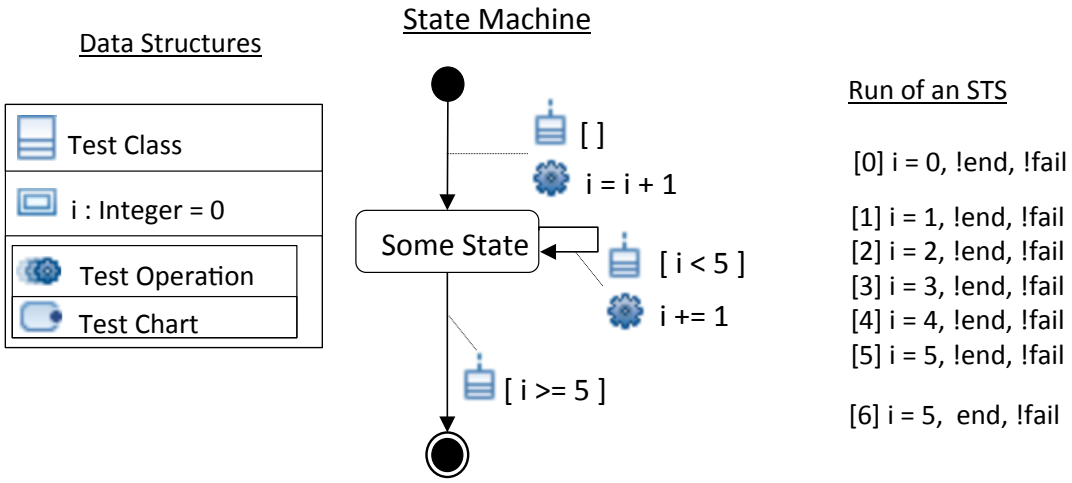


Figure 2.24: RMOF instance example with STS semantics

The syntax and semantics of SPEM as RMOF instance is described in the next chapter.

3 Software Process Engineering Metamodel

> "Design is the method of putting form and content together."
>
> Paul Rand

This chapter introduces the Software Process Engineering Metamodel as RMOF instance. The syntax is presented in the next section 3.1. Since all specifications of the Object Management Group are informal the approach is complemented by adding semantical building blocks for a Petri Net based semantic in terms of RMOF instances in section 3.2. These semantical building blocks are used in section 5.6 on page 180 to analyze the impacts of process model changes.

3.1 Syntax

The OMG defines the Software & Systems Process Engineering Meta-model Specification 2.0 (SPEM [6]) as a Meta-Model to describe Software and System Engineering Processes on a general level focusing on the compliance to various kinds of existing process modeling languages and process models. SPEM separates strictly between definition and application of process models as presented in figure 3.1, supporting a maximal reuse of defined process elements. An example for this separation are, e.g., Role Uses as instances of Role Definitions bound together in the so called "Guidance". SPEM 2.0 has been designed in different packages, namely:

- **Core:** Defines abstract common classes, e.g., "Class", "Association", and "Action" including the ability to create user-defined qualifications to for example distinguish different 'kinds' of SPEM 2.0 class instances.
- **Process Structure:** Defines basic structural elements for defining development processes, e.g., "BreakdownElement" with children like "Milestone" and "WorkProductUse".
- **Process Behavior:** Since SPEM does not provide its own behavior modeling concept this package is used to introduce links to UML2 Superstructure behavior related elements like State Charts and Activity Charts.

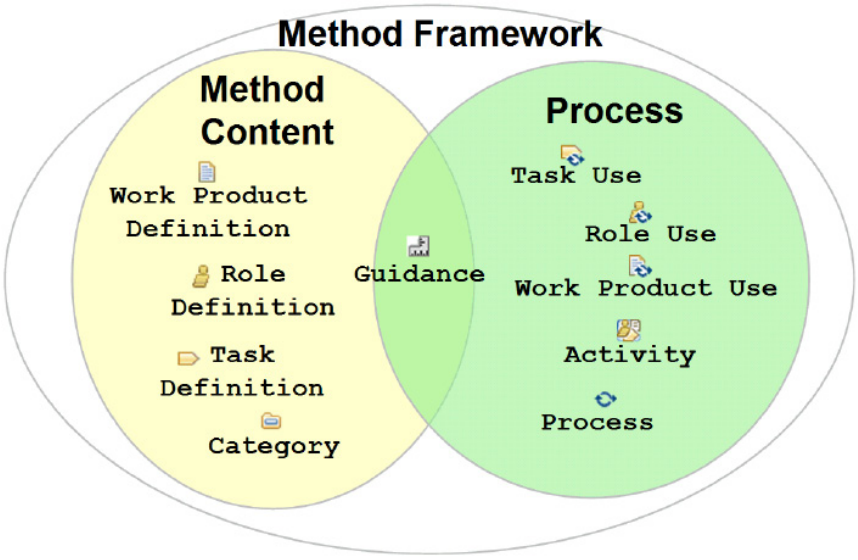


Figure 3.1: SPEM 2.0 method framework

- **Managed Content:** Defines fundamental concepts for managing of textual descriptions for process and method content elements, e.g., "Guidance" and "Metric".
- **Method Content:** Defines core elements of every method such as "Roles", "Tasks", and "Work Product" definitions.
- **Processes with Methods:** separates the reusable core method content from its application in processes this package binds both together again, e.g., "Task Definitions" and "Task Uses".
- **Method Plugin:** defines capabilities to manage libraries of Method Content and Processes.

Figure 3.2 presents the behavioral links of package "Process Behavior". These links will be extended and filled with appropriate RMOF semantics.

SPEM links State Machines and Activity Charts as behavior UML elements. State Machines are used for Work Product Definitions, Activity Charts for Work Definitions. Formalization of State Charts and Activity Charts often assign some kind of finite Automata semantics to State Charts and some kind of Petri Nets for Activity Charts. The Eclipse Process Framework (EPF [86]) is the reference implementation of SPEM 2.0. The modeling framework has been used to describe some example processes such as Open Unified Process (an Open Source Variant of the Rational Unified Process),

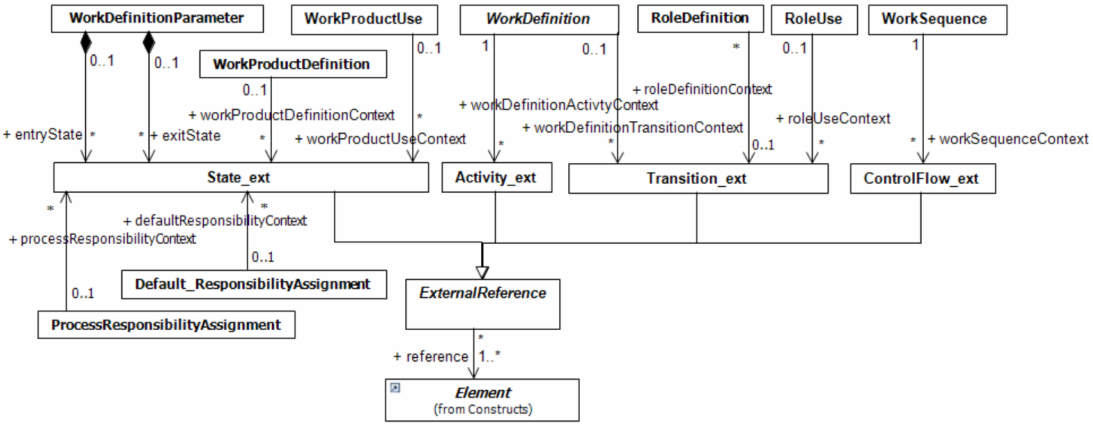


Figure 3.2: Process behavior package of SPEM

Scrum [87], and eXtreme Programming [88]. EPF can mainly be used to publish these processes in form of a web page as illustrated in figure 3.3.

Such a web page is mainly used to describe process elements, e.g., what has to be done in "Requirements Evaluation Step", what Roles are involved or what are predecessors and successors of an activity. But no behavior is executed. The name indicates, that EPF is based on Eclipse and the data model is based on the Eclipse Modeling Framework (see 2.22). The EMF models used in this thesis are based on the EPF implementation. This ensures a syntactical compliance between EPF and the RMOF SPEM instance. More precisely the RMOF SPEM instance is a superset of EPF because extended in several ways e.g., to cover tool interactions or concrete artifacts like Matlab Simulink Models.

SPEM behavior points to Activity Diagrams of UML2, which also do not define formal semantics, but give some hints in the intended semantic direction e.g., to "use a Petri-like semantics instead of State Machines". Nevertheless there exist several scientific papers bridging this gap and a mapping is defined in a more or less straight forward way (compare 3.2). A little bit more complex are additional "features" like (sub-) activity calls, inhibitor and weighted arcs, priorities, self modifying nets, data flows, events, probabilism and time. Instead of defining a single concrete semantics (like in the so called "executable SPEM" approach - defining the semantics of SPEM in terms of Prioritized Time Petri Nets) formal semantic building blocks expressed in RMOF are introduced in the next section. These building blocks are composed and used to simulate the previously identified interaction sequences.

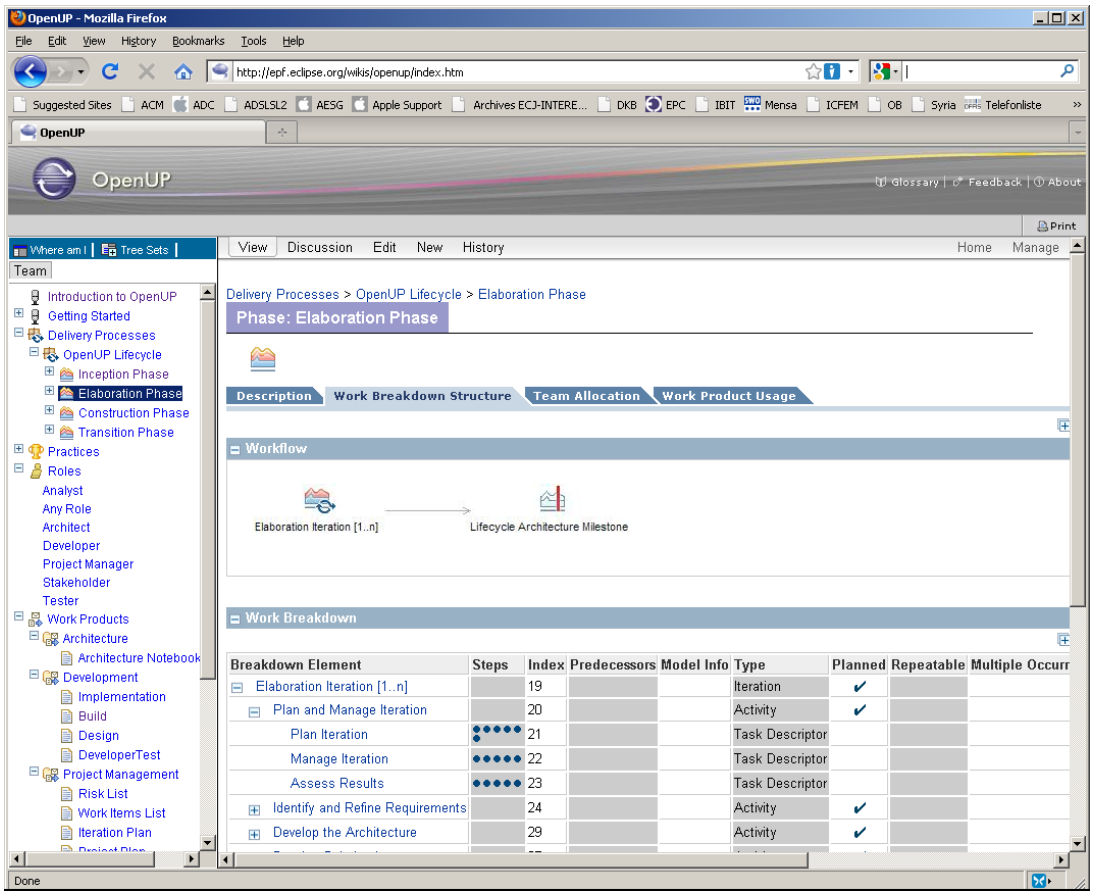


Figure 3.3: Eclipse Process Framework screenshot

3.2 Semantics

Carl Adam Petri invented his Petri Nets at the age of thirteen to describe chemical processes and documented them as part of his dissertation “Communication with Automata”. Petri Nets are widely used in Theory and Practice [89, 90]. They are a very effective way to illustrate concurrency. Standard Place/Transition Petri Nets as RMOF instance are introduced in this section and extended in a so called ’weakly colored’ variant to distinguish different tokens sets running on the same Petri net. The tokens can represent objects and have additional properties like probability and interaction effects (e.g., manually edited test vectors).

Petri Nets [91] can switch concurrently a set of (enabled) transitions in contrast to State Machines. The level of concurrency depends on the number of so called “Tokens”

on each Place (which is equal to State in State Machines). A firing transition consumes tokens on source places and produces tokens on target places. Petri Nets are defined formally:

Definition 27 (Petri Net) *A Petri Net, N is a tuple $(P, T, \mathbb{F}, \mathbb{B})$ with*

- *a finite, ordered set $P = \{p_1, \ldots, p_{|P|}\}$ of Places,*
- *a finite, ordered set $T = \{t_1, \ldots, t_{|T|}\}$ of Transitions,*
- *with $P \cap T = \emptyset$*
- *a $|P| \times |T|$-matrix $\mathbb{F}$ over $\mathbb{N}$ (called forward matrix), and*
- *a $|T| \times |P|$-matrix $\mathbb{B}$ over $\mathbb{N}$ (called backward matrix).*

■

On base of the forward and backward matrices we can define edges.

Definition 28 (Petri Net::Edges) *Let $(P, T, \mathbb{F}, \mathbb{B})$ be a Petri Net. Then $F : P \times T \cup T \times P \mapsto \mathbb{N}$ is the edge function, defined as $\forall x, y \in P \cup T$:*

$$F(x,y) := \begin{cases} \mathbb{F}_{i,j}, & \text{if } x = p_i \wedge y = t_i \\ \mathbb{B}_{i,j}, & \text{if } x = t_i \wedge y = p_i \end{cases}$$

■

The state space of a Petri Net consists of all possible markings (= number of tokens on the places). Mathematically formulated let $N = (P, T, \mathbb{F}, \mathbb{B})$ be a Petri Net, the state space of $N = \mathbb{N}^P$. A mapping $s : P \mapsto N$ is called state or marking of the Petri Net denoting the number of tokens on a place.

Definition 29 (Petri Net::Firing) *Let $N = (P, T, \mathbb{F}, \mathbb{B})$ be a Petri Net, $s \in \mathbb{N}^P$ a state of N and $t \in T$. t is enabled in s if $s \geq \mathbb{F}(t)$. t fires from s to s', if t is enabled in s and $s' = s - \mathbb{F}(t) + \mathbb{B}(t)$.* ■

Figure 3.4 shows an example Petri Net. According to the number of tokens t_1 and t_2 are able to switch concurrently, non-deterministically, and arbitrarily often during a single step. The transition sets $\{\{\}, \{t_1\}, \{t_2\}, \{t_1, t_1\}, \{t_1, t_2\}, \{t_2, t_2\}\}$ can fire in the Petri Net of figure 3.4. Petri Nets of the definitions 27-29 are called Place/Transition or Standard Petri Nets.

Place/Transition Petri Nets are extended by using so called "Weak Colors" differentiating several Place/Transition Petri Nets, e.g., to distinguish different resource

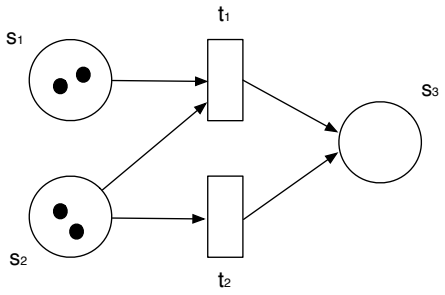


Figure 3.4: Petri Net Example

categories represented by each token set. The following instances are used to instantiate a P/T-Petri Net based switching behavior in RMOF. First, the data structures to handle the different colored token sets are described.

30.1 TokenLibrary = (Class,{(ownedAttribute, (weaklyColoredBags)), (generalization, {Object})}) holds all Petri Net based tokens for a specific context, e.g., an Activity Chart.

- weaklyColoredBags = (Property,{(aggregation, composite), (type, WeaklyColoredBag), (isOrdered, *false*), (lowerValue, 0), (upperValue, *)}) holds all weakly colored bags.
- i = (Property,{(type, Integer) }) counter variable.
- getWeaklyColoredBag = (Operation,{(ownedParameter, (weakColor, weaklyColoredBag)),})
  - weakColor = (Parameter,{(type, Integer)})
  - weaklyColoredBag = (Parameter,{ (direction, return) (type, WeaklyColoredBag)})

  gets a (weakly) colored bag. If the bag exists in the library it is returned, otherwise a new bag is created. The corresponding RMOF State Machine is presented in figure 3.5.

30.2 WeaklyColoredBag = (Class,{(ownedAttribute, (weakColor, bag)), (generalization, {Object})}) holds a single weakly colored bag.

- weakColor = (Property,{ (aggregation, composite), (type, Integer), (lowerValue, 1), (upperValue, 1)}) defines the (weak) color of the weakly colored bag. This corresponds to the type of the objects this bag can hold.
- bag = (Property,{ (aggregation, composite), (type, Marking), (lowerValue, 0), (upperValue, *)}) holds a set of marking for the bag.

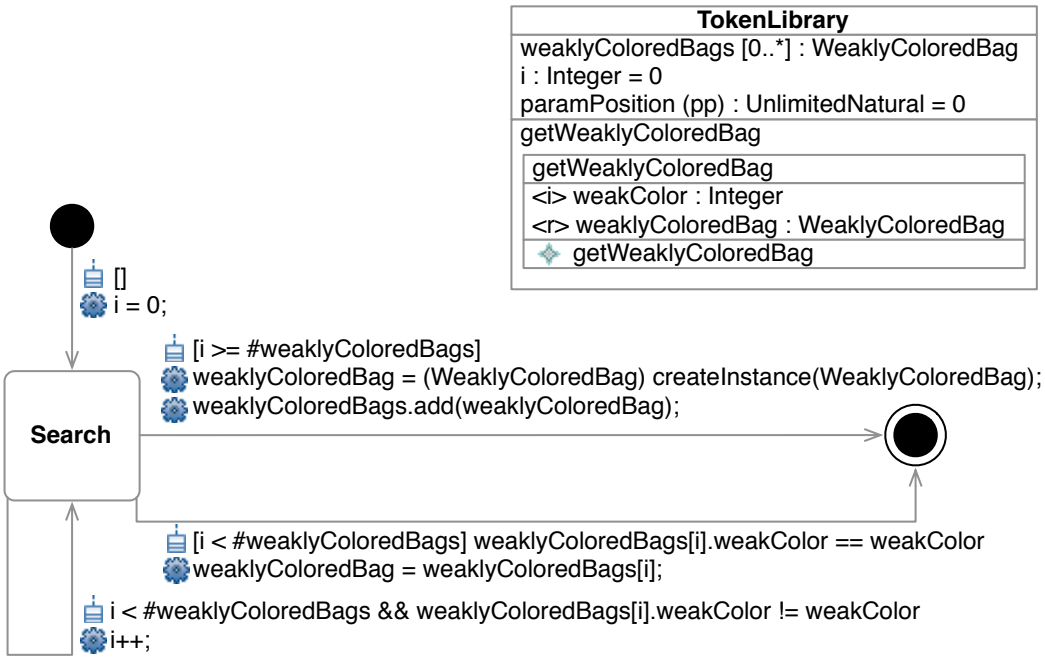


Figure 3.5: *getWeaklyColoredBag* of class *TokenLibrary*

- i = (Property,{(type, Integer) }) counter variable.
- getMarking = (Operation,{(ownedParameter, (place, marking)),})
  - place = (Parameter,{(type, Integer)})
  - marking = (Parameter,{(direction, return) (type, Marking)})

  gets a marking of a (weakly) colored bag. If the marking exists in the bag it is returned, otherwise a new marking is created. The corresponding RMOF State Machine is presented in figure 3.6.

30.3 Marking = (Class,{(ownedAttribute, (place, token)), (ownedOperation, (changeNumberOfTokens)), (generalization, {Object})}) holds a marking.)

- place = (Property,{(aggregation, composite), (type, ObjectID)}) refers to the ObjectID of the place that is holding the tokens.
- token = (Property,{(aggregation, composite), (type, Integer)}) number of tokens available in the place.
- change = (Operation,{(ownedParameter, (tokenDelta)),})
  - tokenDelta = (Parameter,{(type, Integer)})

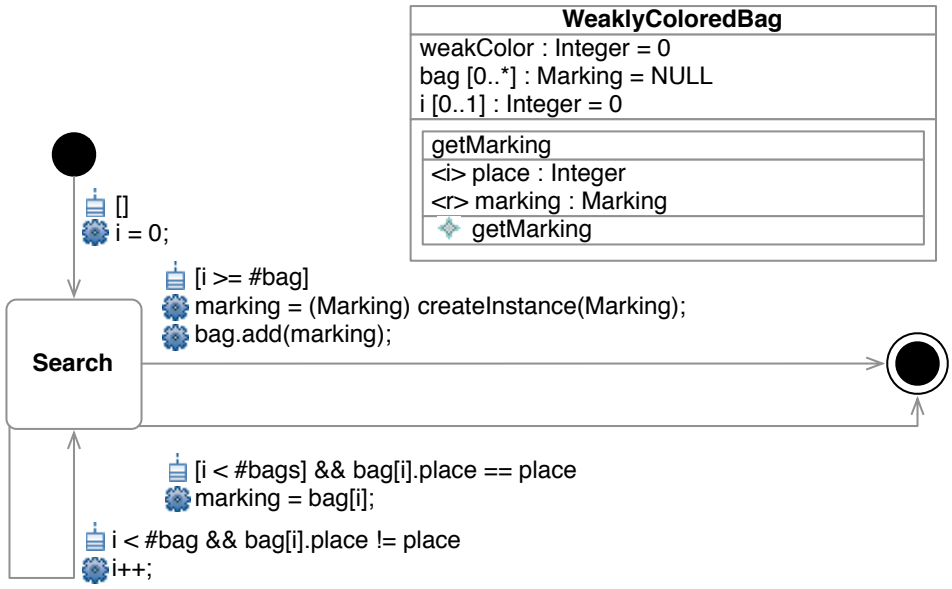


Figure 3.6: *getMarking* of class *WeaklyColoredBag*

adds and removes tokens on the place (if possible) The corresponding RMOF State Machine is presented in figure 3.7.

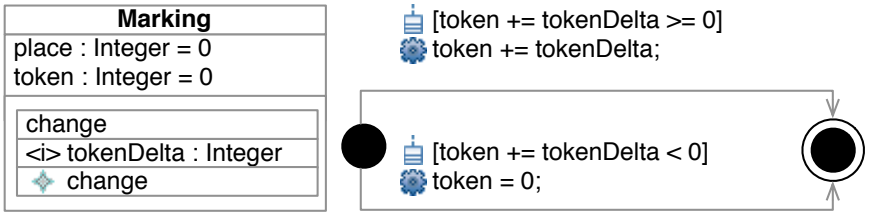


Figure 3.7: *change* of class *Marking*

After the required sources and sinks for tokens were introduced, the required firing behavior of Petri Nets is added.

(ii) Relevant - determines how many times a transition could switch (locally) - in relation to a subset of weakly colored markings. It is further required to distinguish different (weak) token colors.

31.1 TransitionLibrary = (Class,{(ownedAttribute, (weaklyColoredTransitions)), (generalization, {Object})}) holds all Petri Net based transitions for a specific context (e.g., activity chart).

- weaklyColoredTransitions = (Property,{ (aggregation, composite), (type, WeaklyColoredTransitions), (lowerValue, 0), (upperValue, *)}) holds all weakly colored bags.

- i = (Property,{(type, Integer) }) counter variable.

- switchingTogether = (Property,{ (aggregation, composite), (type, SwitchingTogether), (lowerValue, 0), (upperValue, *)}) holds a set of all sets (of colors) that switch together.

- computeSwitchingTogether = (Operation,{ (ownedParameter, ()),}) Computes the colored set that switch together. The corresponding RMOF State Machine is presented in figure 3.8.

31.2 SwitchingTogether = (Class,{(ownedAttribute, (together)), (generalization, {Object})}) defines a set of colors that is switched together.).

- switchingTogether = (Property,{ (aggregation, composite), (type, Integer), (lowerValue, 0), (upperValue, *)}) holds a set of colors.

31.3 WeaklyColoredTransitions = (Class,{ (ownedAttribute, (i, weakColor, weaklyColoredTransition)), (generalization, {Object})}) holds a single weakly colored set of transitions.)

- i = (Property,{(type, Integer) }) counter variable.

- weakColor = (Property,{ (aggregation, composite), (type, Integer), (lowerValue, 1), (upperValue, 1)}) defines the (weak) color of the weakly colored set of transitions.

- transitions = (Property,{ (aggregation, composite), (type, WeaklyColoredTransition), (isOrdered, *true*), (lowerValue, 0), (upperValue, *)}) holds a set of transitions.

- relevances = (Property,{ (aggregation, composite), (type, Integer), (isOrdered, *true*), (lowerValue, 0), (upperValue, *)}) holds a set of transitions.

- getPNTransition = (Operation,{ (ownedParameter, (id, pnTransition)),})
  - place = (Parameter,{ (type, Integer)})
  - marking = (Parameter,{ (direction, return) (type, PNTransition)})

  gets a Petri Net transition of a (weakly) colored set of transitions. If the transition with the object ID exists in the set of transitions it is returned, otherwise a transition.

- computeRelevance = (Operation,{(ownedParameter, (weakColor, relevance)),})

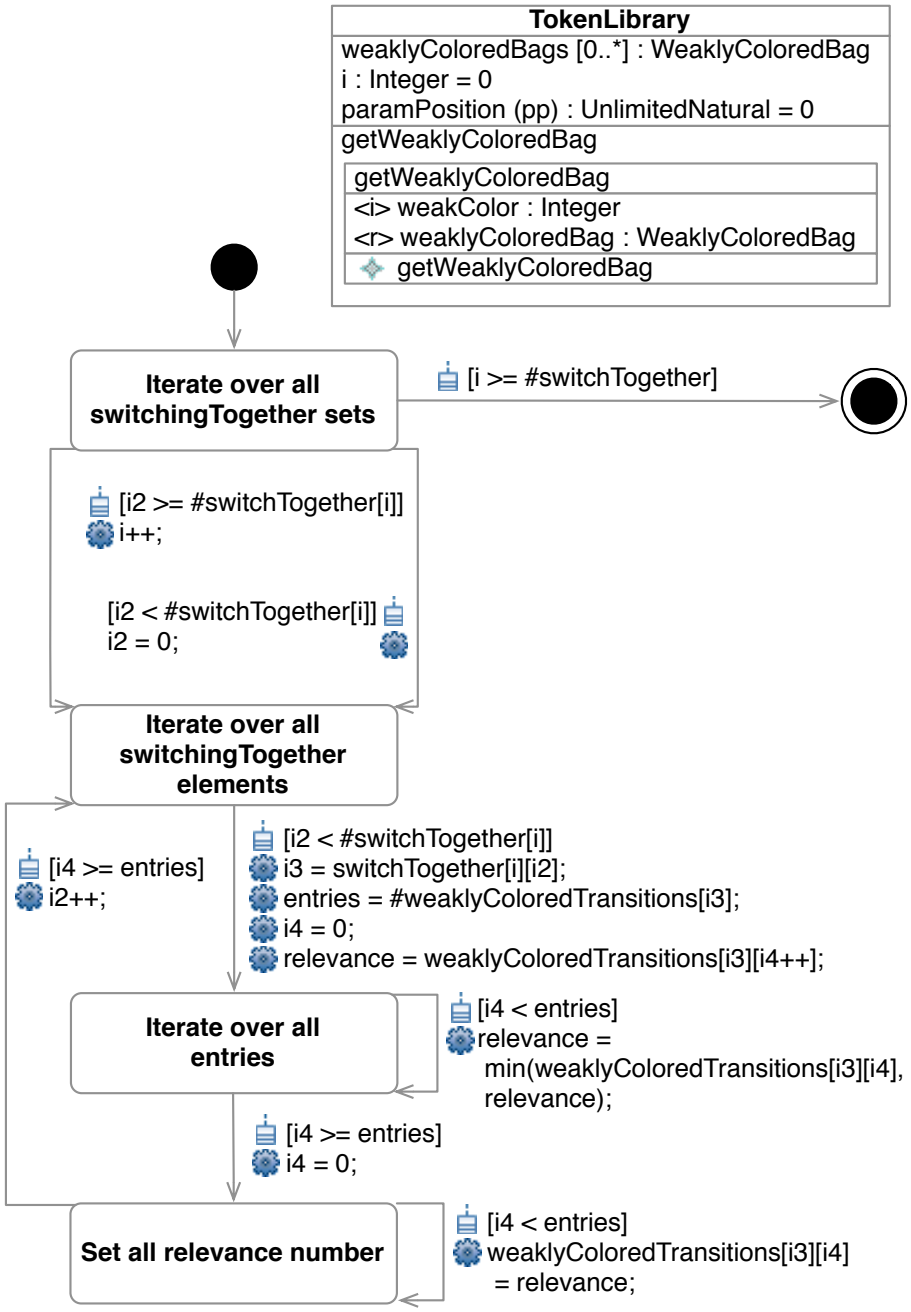


Figure 3.8: *computeSwitchingTogether* of class *TransitionLibrary*

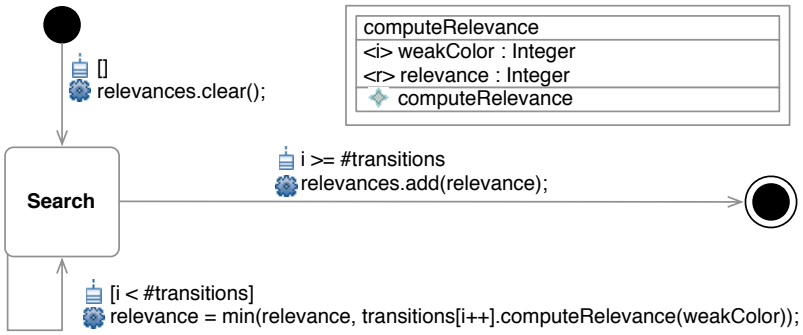


Figure 3.9: *computeRelevance* of class *WeaklyColoredTransitions*

- weakColor = (Parameter,{(type, Integer)})
- relevance = (Parameter,{(direction, return) (type, Integer)})

Computes the relevance of a weakly colored set of transitions with respect to a (weak) color. The corresponding RMOF State Machine is presented in figure 3.9.

- computeRequiredTokens = (Operation,{(ownedParameter, (weaklyColoredTransitionInput, weaklyColoredBag)),})
  - weaklyColoredBag = (Parameter,{ (direction, return) (type, WeaklyColoredBag)})

  Computes the required tokens of a weakly colored bag. The corresponding RMOF State Machine is presented in figure 3.10.

- computeAvailableTokens = (Operation,{(ownedParameter, (weaklyColoredBag)),})
  - weaklyColoredBag = (Parameter,{ (direction, return) (type, WeaklyColoredBag)})

  Computes the available tokens of a weakly colored bag. The corresponding RMOF State Machine is presented in figure 3.11.

- computeSubBagOf = (Operation,{(ownedParameter, (weaklyColoredTransitionsInput, weaklyColoredTransitions)),})
  - weaklyColoredTransitionsInput = (Parameter,{(direction, input) (type, WeaklyColoredTransitions)})
  - weaklyColoredTransitions = (Parameter,{(direction, return) (type, WeaklyColoredTransitions)})

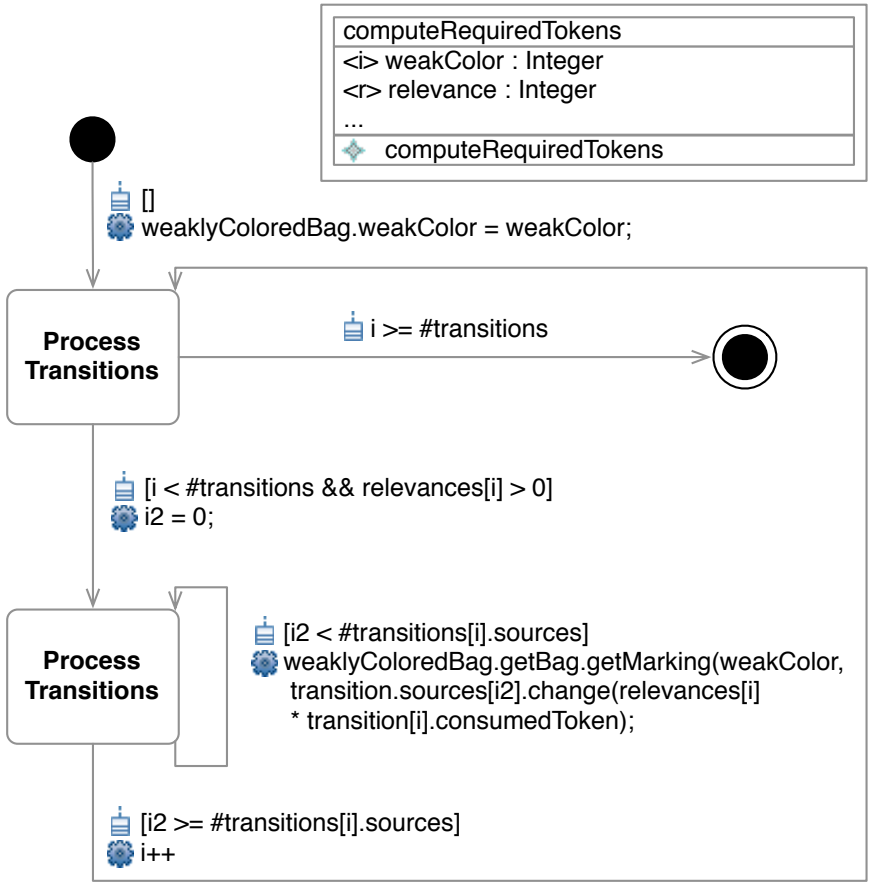


Figure 3.10: *computeRequiredTokens* of class *WeaklyColoredTransitions*

Computes a subbag of weakly colored Transitions. The corresponding RMOF State Machine is presented in figure 3.12.

- computerMaximality = (Operation,{(ownedParameter, (weaklyColoredTransitionsInput, weaklyColoredTransitionsMax)),})
  - weaklyColoredTransitionsInput = (Parameter,{(direction, inout) (type, WeaklyColoredTransitions)})
  - weaklyColoredTransitions = (Parameter,{(direction, input) (type, WeaklyColoredTransitions)})

  Computes maximal subbag of weakly colored Transitions. The corresponding RMOF State Machine is presented in figure 3.13.

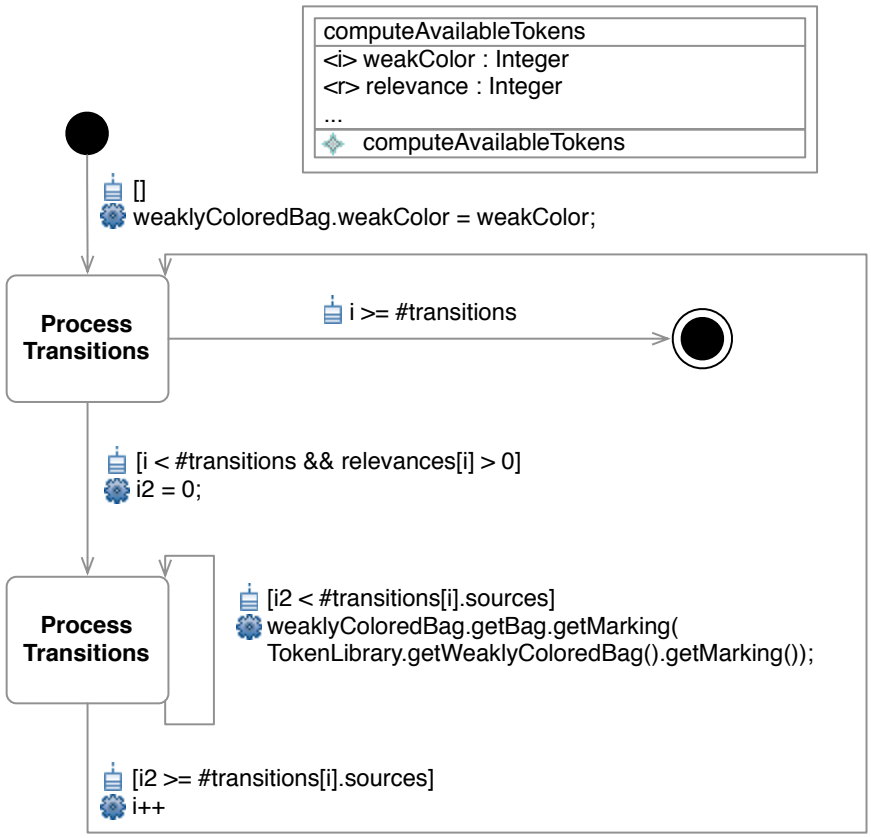


Figure 3.11: *computeAvailableTokens* of class *WeaklyColoredTransitions*

31.4 PetriNetTransition = (Class,{(ownedAttribute, (i, localRelevance, sources, targets, consumingToken, producingTokens)), (ownedOperation, (computeRelevance)), (generalization, {Object})}) holds the dynamic related aspects of a Petri Net based transition.

- i = (Property,{(aggregation, composite), (type, Integer)}) counter variable.
- localRelevance = (Property,{ (aggregation, composite), (type, Integer)}) stores a computed relevance.
- sources = (Property,{ (aggregation, composite), (upperValue, *), (type, Marking)}) refers to sources of the transition.
- targets = (Property,{(aggregation, composite), (upperValue, *), (type, Marking)}) refers to the targets of the transition.
- consumingTokens = (Property,{(aggregation, composite), (type, Integer)})

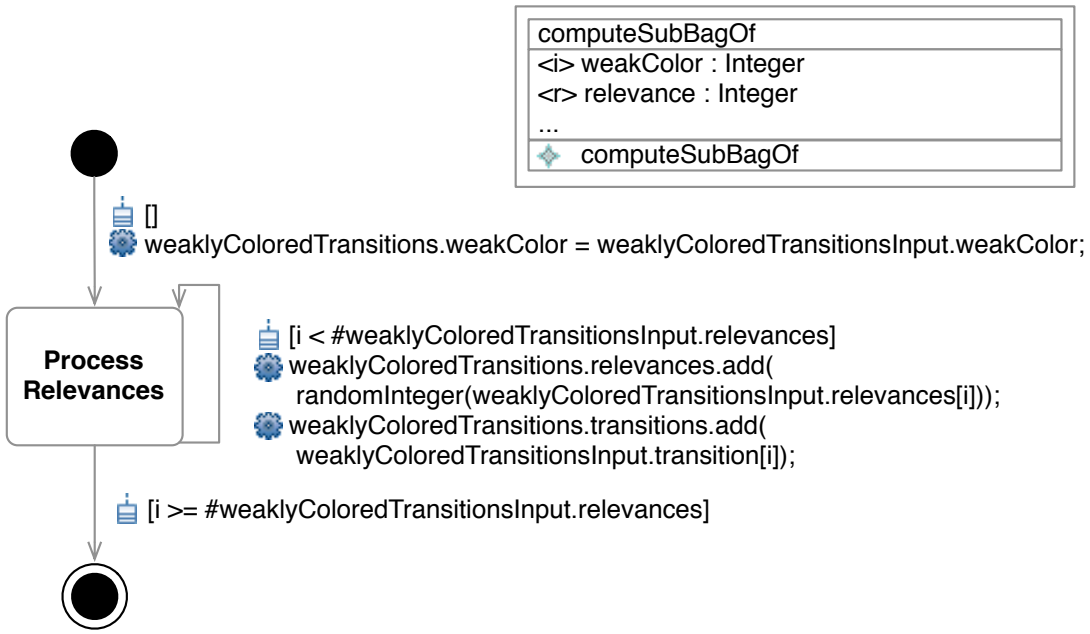


Figure 3.12: *computeSubBagOf* of class *WeaklyColoredTransitions*

the transition consumes this number of tokens on all sources when it fires.

- producingTokens = (Property,{(aggregation, composite), (type, Integer)}) the transition produces this number of tokens on all targets when it files.

- computeRelevance = (Operation,{(ownedParameter, (weakColor, relevance)),})
  - weakColor = (Parameter,{(type, Integer)})
  - relevance = (Parameter,{(direction, return) (type, Integer)})

  Computes the relevance with respect to a (weak) color. The corresponding RMOF State Machine is presented in figure 3.14.

(iii) Enabled - determining if the transition is enabled. In a first approach this equals to a transition guard that is evaluated to *true*. The enabled frame operation is introduced in the class

31.1-1 TransitionLibrary2 = (Class,{ (ownedAttribute, (weaklyColoredTransitions)), (ownedOperation, (computeEnabledTransitions)), (generalization, {Object})}) holds all Petri Net based transitions for a specific context (e.g., activity chart).

- computeEnabledTransitions = (Operation,{}) Invokes all filters of not enabled transitions. The corresponding RMOF State Machine is presented in

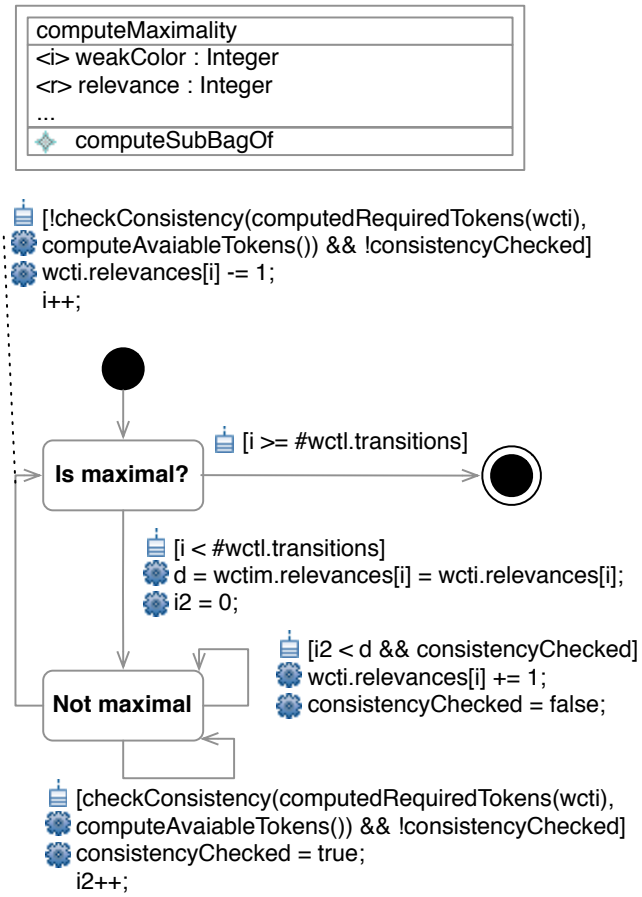


Figure 3.13: *computeMaximality* of class *WeaklyColoredTransitions*

figure 3.15. Since there might be arbitrary conditions that describe what is enabled and what not there is an extensible enable filter pipeline (concatenated in form of a conjunction). All filter classes names contain "enableFilter" and are applied if they have a property "isActive" of type "Boolean" that evaluates to *true*.

The first class simply checks, if the transition guard evaluates to *true*.

31.5 enableFilter-Guards = (Class,{(ownedAttribute, (isActive)), (ownedOperation, (filter)), (generalization, {Object})})

- isActive = (Property,{(aggregation, composite), (type, Boolean)}) specifies if the filter is active or not.
- filter = (Operation,{ (ownedParameter, (transitionsInput, transitionsOut-

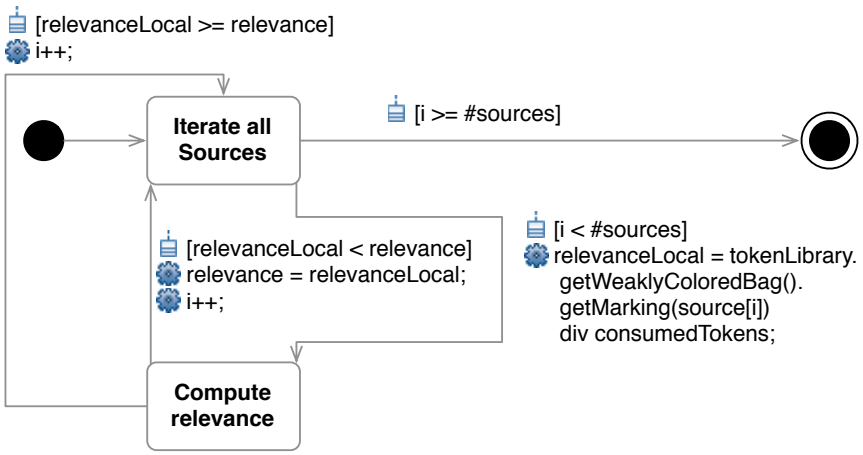


Figure 3.14: *computeRelevance* of class *PetriNetTransition*

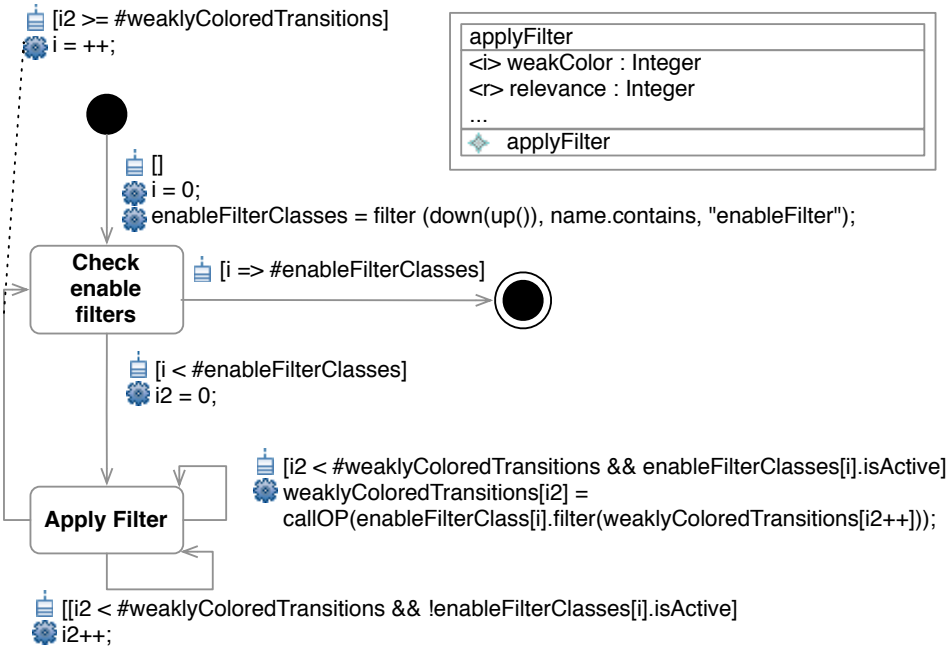


Figure 3.15: *computeEnabledTransitions* of class *TransitionLibrary*

put)),})

- transitionsInput = (Parameter,{(isUnique, false), (upperValue, *), (type, weaklyColoredTransition)})

- transitionsOutput = (Parameter,{ (isUnique, false), (upperValue, *), (direction, return) (type, weaklyColoredTransition)})

filters all transitions whose guard evaluates to *true*. The corresponding RMOF State Machine is presented in figure 3.16.

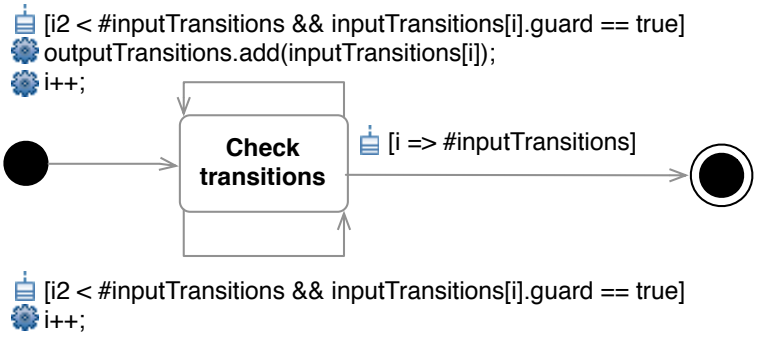


Figure 3.16: *filter* of class $enableFilter - Guards$

(iv) Consistent - determining a subbag of transitions that can switch concurrently. This is simply realized by computing a (randomized) subbag and checking the subbag relationships.

(v) (Maximal) - ensuring that the (non-deterministically) subset is maximal. This is realized by adding as many transitions to the bag of transitions that defines the firing as possible (staying consistent).

In order to map objects to tokens it is necessary to introduce an additional attribute in each object "isWeaklyColoredToken" of type "Boolean". This can be used for the

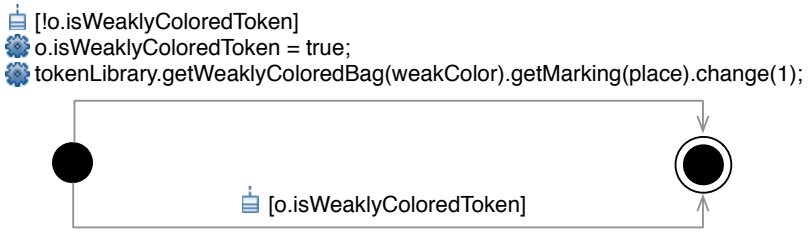


Figure 3.17: Introduce objects as tokens

transfer as presented in figure 3.17. *o* is the object that should be presented as token with the weak color *weakColor* on the marking with the place *place*.

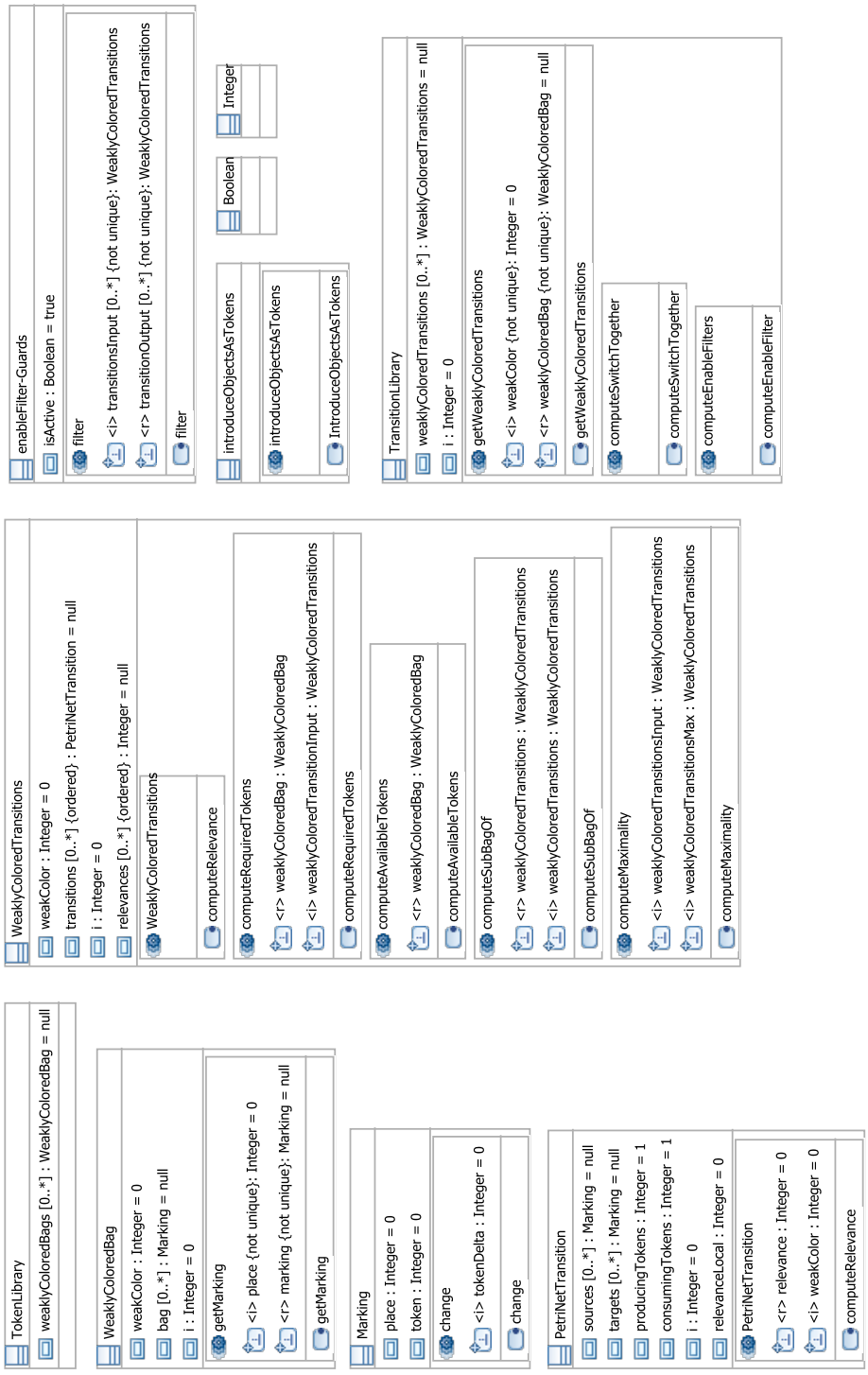


Figure 3.18: Classes of weakly colored, object-based Petri Net library

Figure 3.18 shows a class diagram of the previously defined semantical building blocks to define a weakly colored, object-based Petri Net based transition library. The analysis and in particular the synthesis of RMOF based models is presented in the next chapter based on metaheuristics as discussed in section 1.4.3 exploiting the symbiotic relationship between RMOF and metaheuristics and avoiding the restrictions of other methods.

4 Synthesis and Analysis

> "The purpose of computing is insight, not numbers."
>
> R. Hamming

This chapter focusses on the synthesis of RMOF based models including some synthesis dependent analysis tasks e.g., to evaluate the synthesized models. Another (application depended) analysis is presented in section 5.6 of the evaluation. The approach is based on Genetic Programming which is introduced in section 4.1 including a short discussion of the computation complexity in general in section 4.1.1, which is one reason to apply these methods and techniques. The second reason is that Genetic Programming matches very well the RMOF language pyramid. This aspect is dicussed in section 4.2 and concluded with introduced optimizations in section 4.3.

4.1 Genetic Programming

Process model optimization includes two tasks: synthesis of process model (parts) and process model analysis. As discussed in section 1.4 for both problems methods exist that can be classified in different categories [92]:

(i) Enumerate methods search the complete problem space successively. This is often not possible when facing hard complexity problems. An enumerate method is e.g., Model Checking [51]. There was considerable progress in the last years regarding the problem space that can be checked with Model Checking by using efficient representations, abstractions and over-approximations [1]. A naive application is still often not possible in particular when the problem space is not well known / understood. It should always be possible to define a termination criteria before the complete space is checked but the search ordering is often not optimized to find (good) approximates.

(ii) Calculus-based methods apply mathematical functions and algorithms in a deterministic way. This requires that the calculus matches relevant properties of the search space appropriately. In contrast figure 4.1 shows a scatter plot of data without a linear, obvious mathematical structure. A naive application is often inefficient in particular when the problem space is not well known / understood.

(iii) Random methods apply stochastic optimization methods. A stochastic optimization can be effective if the problem space is unknown or not well understood to efficiently find sub optimal / partial solutions. The prerequisite of this method is a kind of continuous evaluation of partial solutions meaning that a small input change will likely produce a small effect evaluation.

The three different method categories are inversely sorted by the degree of knowledge of the problem space and in relation to their computation complexity. If the problem space is well understood including the effects of the algorithms to solve the problem it is possible to use a previous method category. In this thesis a hybrid approach is used

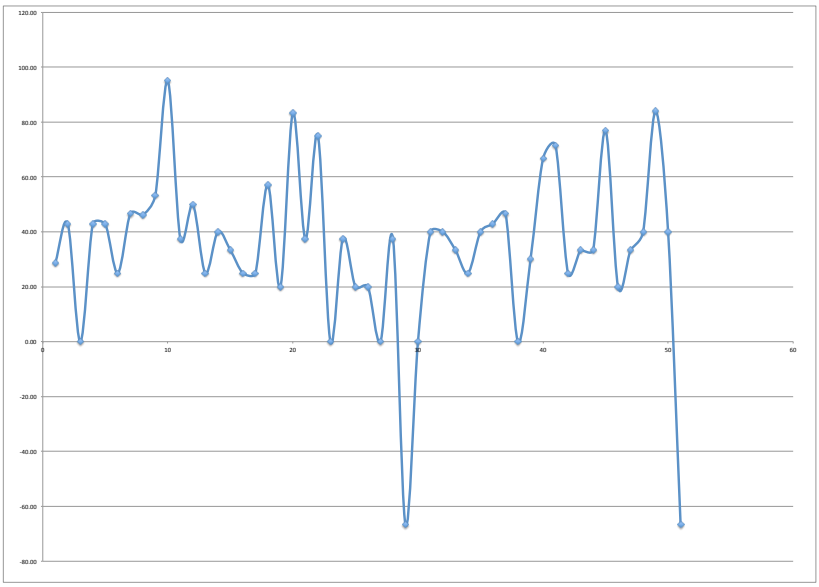


Figure 4.1: Scatter plot without (obvious) mathematical structure

to optimize process models. Methods and techniques from the field of stochastic optimization are used to synthesize process models because the methods can be applied to a wide range of problems, they fit the problem space (learning relationships, models, meta-models) and the problem space is inherently unrestricted. This approach also matches the RMOF meta language approach with its flexibility to cover all kinds of language elements. The long term goal is to move to calculus or enumerative methods when the problem space is well understood. The analysis of the models is enumerative e.g., to check the quality of a synthesized process model. Fortunately the problem space can be reduced to tracked interaction sequences as shown in the evaluation section 5 instead of facing all possible interaction sequences. In this section the synthesis of process models (or RMOF instances in general) with stochastic optimization, meta-

heuristics, and in particular genetic programming will be discussed.

"Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems. Metaheuristics are the most general of these kinds of algorithms, and are applied to a very wide range of problems. [...] Metaheuristics are applied to 'I know it when I see' it problems. The algorithms used to find answers to problems when you have very little to help you: you do know what the optimal solution looks like, you don't know how to go about finding it in a principled way, you have very little heuristic information to go on, and brute-force search is out of the question because the space is too large. But if you're given a candidate solution to your problem, you can test it and assess how good it is. That is, you know a good one when you see it." ([93], page 9)

Metaheuristics is the primary subfield of stochastic optimization that is used in this thesis. A general assumption in this field is that similar solutions tend to behave similarly (and tend to have similar quality), so small modifications will generally result in small, well-behaved changes in quality, allowing us to "climb the hill" of quality up to good solutions. A genetic algorithm (GA) is a metaheuristic that mimics the process of natural evolution. A natural evolution is conducted by a Darwinian evolutionary system. Such a system embodies at least one population of individuals competing for limited resources, the concept of birth and death of individuals, a concept of fitness which reflects the ability to survive and reproduce, and a concept of variational inheritance meaning that offspring closely resembles their parents but are not identical [94].

In 1960 Ingo Rechenberg developed a set of algorithms known as the Evolution Strategies (ES [95]). The simplest ES algorithm is the (μ, λ) algorithm. λ is the number of random individuals constituting the initial population (and the number of individuals in total of every evolution step). In each evolution step the complete population is deleted but the μ fittest ones. The remaining μ individuals are used to produce new λ individuals, whereby the children substitute their parents. The production is done by using mutation and crossover operations. The different ES strategies vary mainly in the production rules, e.g., in another strategy, called $\mu + \lambda$, the parents are only substituted by their offspring if the offspring is fitter than the parents. Fogel [96] developed so called Evolutionary Programming (EP) by extending the strategies to different representations, in particular finite state automata. Thus, he changed the production rules to add, change, delete nodes and/or transitions.

Holland was the first who introduced a formal framework 1975 [97] describing Genetic Algorithms consisting of the following elements:

- $A = \{A_1, A_2, \cdots\}$: the structures/system undergoing the adaptation.

- E : an environment which sends (on a discrete time scale t) input signals to the system. The input history is written as $< I(1), I(2), \cdots, I(t-1) >$.
- M : a memory storing parts of the input history.
- λ : the adaptive plan which determines successive structural modifications in response to the environment, defined $\mu : I \times (a \times M) \mapsto (a \times M)$. Often adaptive plan applies operators $\Omega = \{\omega : A \mapsto P\}$, where P is a probability distribution.
- μ : a measure of performance of the system(s) defined as $\mu_E : A \mapsto \mathbb{R}$.

Holland defines a cumulative payoff function $U_{\lambda,E}(T) = \sum_{t=1}^{T} \mu_E(A(\lambda, t))$. The performance target can be formulated in terms of the greatest possible cumulative payoff in the first T time-steps:

$$U_E^*(T) = lub_{\lambda \in \Lambda} U_{\lambda,E}(T)$$

The mean fitness is defined as

$$\overline{\mu}(\tau, t) = \frac{\sum_{A \in a_\tau(t) \mu_E(A)}}{P(\tau, t)}$$

$P(\tau, t)$ is the number of individuals in $a_\tau(t)$. A robustness criteria defined as

$$\underset{E \in \varepsilon}{glb}\, \underset{t}{glb}\, \underset{\lambda' \in J}{glb}\, \overline{\mu}_E(\lambda, t) / \overline{\mu}_E(\lambda', t)$$

Holland divides A into subsets to discuss the optimal allocation of trials based on the total expected loss of fitness for any allocation of n trials. Figure 4.2 visualizes the idea (compare [97], page 67, figure 9). The subsets base on so called schemata (or similarity template). Each schema $(V1, V2)$ is specified as a kind of template with "don't care" positions. For example: Let $A_i \in \{0, 1\}, n \in \mathbb{N}$. Then V1 could be defined as "0010010□010□0 . . .", with □ indicating "don't care" values. An optimal allocation of trials based on the expected loss of fitness (of n trials) is then based on the possible sources of loss. Assuming there are two schemata $V1, V2$ and the algorithm has to decide where to locate new trials two possible sources of loss are faced:

- The observed best $V1$ is really second best, whence $N - n$ trials given $V1$ incur an (expected) individual loss of $|\mu_{V1} - \mu_{V2}|$ which occurs with a probability $q(N - n, n)$.
- The observed best is in fact the best, where the n trials given $V2$ incur the same individual loss $|\mu_{V1} - \mu_{V2}|$ with a probability $1 - q(N - n, n)$.

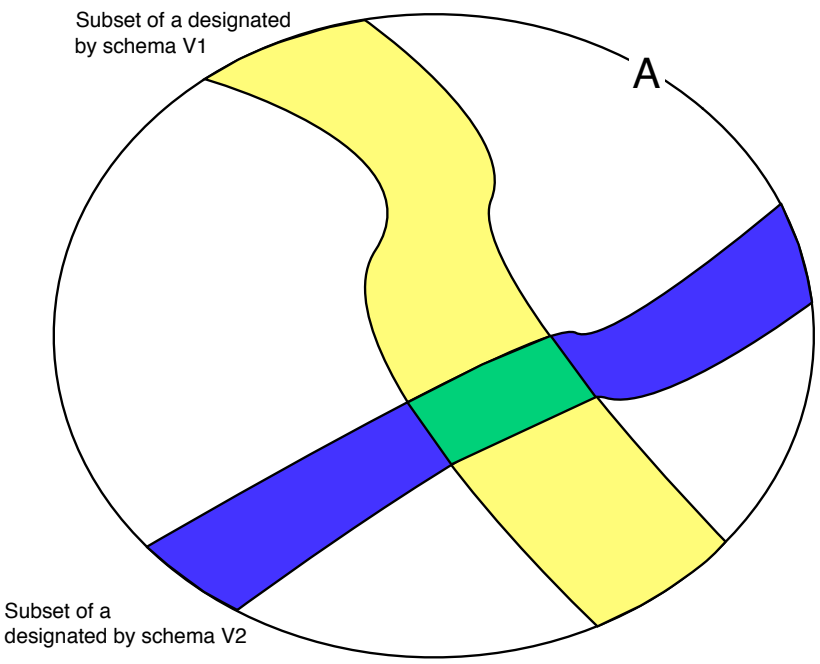


Figure 4.2: GP schemata

Holland defines the total loss for any allocation of n trials to $V2$ and $N-n$ trials to $V1$ as:

$$L(N-n,n) = [q(N-n,n) * (N-n) + (1-q(N-n,n)) * n] * |\mu_{V1} - \mu_{V2}|$$

The schema theorem works for fixed length GA and says that good schemata tend to assist in solving the problem because they will tend to multiply exponentially as the genetic process progresses. Koza [98] adopted the schemata for trees in the subfield Genetic Programming by defining a set of sub-trees as schema. Genetic Programming is a specialization of Genetic Algorithms used to find computer programs. He claimed that the schema theorem also works for variable length GP. O'Reilly and Oppacher added Hollands "don't care" operator to Koza schemata adaptation and investigated the probability of disruption formally. They showed that the maximum probability of disruption varies with size and this is a problem because of the variable length of the GP representation. Altenberg [99] says that crossover was the result of the evolution of evolvability and Banzhaf et al. [100] compares crossover of natural processes with crossover of Genetic Programming with the following differences:

- Function: In biology, the different alleles of swapped genes make only minor changes in the same basic functions. In GP any sub-tree can be changed with another sub-tree (of the same type if strongly typed).

- Context: In GP the context is irrelevant in biology not.
- Disruption: Disruption is prevented in biological systems and increases with the size in GP.

Disruption is prevented when using strong Genetic Types. Banzhaf et al. suggest to improve the crossover operation on base of the measurement of structural and functional similarities. The structural measurement is done by using so called "edit" distances [101] and minimizing these distances during the crossover. The functional measurement is done by evaluating the fitness of the sub-trees.

Genetic Programming uses trees as representation form. The variables and constants in the program are leaves of the tree. In GP they are called terminals, whilst the arithmetic operations are internal nodes called functions. The sets of allowed functions and terminals together form the primitive set of a GP system. The general idea of a GP process is presented in the following listing:

"Steepest Ascent Hill-Climbing With Replacement"

```
n := number of tweaks desired to sample the gradient
S := some initial candidate solution
repeat
  R := Tweak(Copy(S))
  for n-1 times do
    W := Tweak(Copy(S))
    if Quality(W) > Quality(R) then
      R := W
  if Quality(S) > Quality(Best) then
    S := R
until S is the ideal solution or we have run out of time
return S
```

In particular the "Tweaking"/"Copying" and the "Quality" assessment are of interest. In general Tweaking is done in several forms [100]. "Blind Search" neglects the information about previous search results. "Hill Climbing" takes only a new solution as base if the solution is better than the previous found. A special variant is the so called "Simulated Annealing" [102], which is based on an analogy with the cooling of metal in a process of annealing. But this variant is only possible if the optimal fitness is known. The tweaking distance between two GP steps is proportional related to the distance of the optimal solution. "Beam Search" investigates previous promising search points and generates and evaluation metric upon it for new candidates.

Genetic Programming mainly uses lists and trees (as a basic form) to represent individuals (compare [103], [98]). There exist other forms e.g., cellular encoding [104] using a tree to structure operations applied to the genetic program. A simple tree example of the formula $cos(x - sin(x))$ is presented in figure 4.3. In a tree, all non-leaf

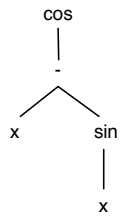


Figure 4.3: GP tree of formula $cos(x - sin(x))$

nodes are non-terminals (i.e. functions) and all leaf nodes are terminals (i.e. values or variables). GP can also be used on forest (= set of trees). It is often required that Genetic Programming is strongly typed [105], meaning that all nodes have types assigned. Types are implemented as atomic types (two types match if they refer the same static unique element), set types (two types match if their static subsets are not-empty), and polymorphic types (two types match if their dynamic subsets are not-empty). Dynamic elements can be created during run-time (of the fitness evaluation) in contrast to static elements. Since GP is per se unrestricted it is often useful to introduce a maximal depth as in the following listing illustrated:

"GP Node growing algorithm"

```
procedure DoGrow(depth, max, Functions)
  if depth >= max then
    return Copy(randomly-chosen leaf node from Functions)
  else
    n := Copy(randomly-chosen node from the Functions)
    l := number of child nodes expected for n
    for i from 1 to l do
      Child i of n DoGrow(depth + 1, max, Functions)
    return n
```

Growing of non-maximal nodes often copies only from non-leaf nodes to ensure growing. Values are often built using so called Ephemeral Random Constants (ERC). ERCs are nodes with GP functions for efficient mutation. Normally the computation of the constants is done in the last possible step of the Breeding (see 4.1.2, page 144) phase because of the computational complexity. The missing parts of Genetic Programming (GP) are now the Breeding step including mutation and crossover as well as the fitness

evaluation.

Before a node is mutated or two nodes are crossed over they are selected. The selection can be done in several ways. An individual i is selected with the probability $p_i = \frac{f_i}{\sum_i f_i}$, where f_i is the fitness of individual i in a "Fitness-Proportional" selection. This selection was applied a long time in the Genetic Algorithm community [97]. Two simple restrictions to μ parents and λ children were introduced by Schwefel 1995 [106] (called (μ, λ) selection), whereby the new μ parents are a subset of the previous λ children. "Ranking Selection" [107] is based on the fitness rank, into which the individuals can be sorted. For a linear ranking

$$p_i = \frac{1}{N}[p^- + (p^+ p^-)\frac{i-1}{N-1}]$$

where $\frac{p^-}{N}$ is the probability of the worst individual being selected, and $\frac{p^+}{N}$ is the probability of the best individual being selected. "Tournament" selection is based on a subset of the population. The size of this subset is called tournament size and defines the selection pressure. The subset is randomly selected. Then a competition takes place, whereby the better individuals replace the worse individuals with their children. Tournament selection has become popular recently because it does not require a fitness comparison between all individuals.

Mutation can be done in several ways, e.g., sub-tree mutation with the algorithm above or mutate an ERC of a random node that has/is an ERC. A special mutation variant is a copying or swapping of sub-tree nodes. If there are two nodes involved this is called crossover. Often the last two choices are preferred because the results are more homologous. When sub-trees are replicated modularity is exploited. This effect perpetuated by so called "Automatic Defined Functions" [108].

In both cases, mutation and crossover, there might be some changes that are not allowed due to constraints regarding syntax or semantics of the final genetic program. All constraints can be implemented as "hard" restrictions directly in the breeding process or as soft restrictions indirectly by reducing the evaluated fitness of the individual. In particular dynamic restrictions are often "soft" to avoid redundant computations in the breeding stage.

The key issue of a GA/GP optimization is to identify and capture computationally useful aspects of evolutionary processes. As a consequence most GA utilize what appear to be rather simplistic assumptions, e.g., a fixed size population, random mating, and static fitness landscapes [94]. The next section discusses the computation complexity of model synthesis in general and in special relationship to some application data.

4.1.1 Complexity

> "The only way of discovering the limits of the possible is to venture a little way past them into the impossible."
>
> A.C. Clarke

The primary reason to use stochastic optimization and in particular Genetic Programming (instead of enumerative methods) concerns the computational complexity which is directly related to the size of the search space. Combinatorics is the mathematical basis for calculating the search space size. Combinatorics define in how many different ways a set of discrete elements can be ordered with respect to arbitrary or ordered selections or repetition of elements. Let s be a set with n elements.

Definition 30 *The permutations of s is a sequence containing all possible, unique subsets of s. The number of permutations of s is defined $P(s) \stackrel{df}{=} n! := n * (n-1) * (n-2) \ldots 1$.*

A combination of order k is considered as a subset of k elements of s.

Definition 31 *The number of different combinations without replacement and order of selection is considered*

$$C_n^k = \binom{n}{k} \stackrel{df}{=} \frac{n!}{(n-k)!k!}$$

called the binomial coefficient.

A variation of order k is considered as a sequence of k elements of s.

Definition 32 *No repetitions allowed:*

$$V_n^k \stackrel{df}{=} \frac{n!}{(n-k)!}$$

Repetitions allowed:

$$\overline{C}_n^k \stackrel{df}{=} \binom{n+k-1}{k}$$

The probability p_A of an event A as outcome of a random experiment is $p_A \stackrel{df}{=} lim_{N \mapsto \infty} \frac{K}{N}$. Additionally the expectation value is defined as follows:

$E(A) = \sum_{y=0}^{\infty} p_A A$ as well as the variance $var(A) = \sum_{y=0}^{\infty} p_A (A - E(A))^2$

The GP complexity has several dimensions:

- GP algorithm complexity to generate random numbers
- Static complexity to generate the structure (e.g., number of nodes) which is a spacial complexity
- Dynamic complexity during fitness evaluation by executing the generated program (often problem depend).

The elements of the GP algorithm are introduced in more detail in the next section.

4.1.2 Algorithm

This section sketches exemplary different elements of a Genetic Programming algorithm described in [93] and implemented in ECJ[1].

Breeding

The initial population and new (sub-)elements of the GP individual are build with the presented grown algorithm (see page 141). A variant of this algorithm builds only full grown individuals (individuals with maximal depth) or switches between the two variants with a certain probability.

Parameters of all variants are the number of retries to avoid redundancies, the minimal and maximal grow depth. Common parameters of the breeding algorithms handle the probabilities to select nodes. For example the probabilities to select terminals (leaves in the GP tree), non-terminals and root nodes. Other probabilities optimize with respect to the evaluation data. For example the breeding algorithm selects nodes with a probability reflecting the relative probability of this node category in the tracking database or selects nodes according to feedback information of the GP evaluation (see section 4.3).

Selection

The simplest form of selection is the random selection, which picks - hence the name - individuals randomly of the previous generation. It is likely that fitter individuals produce fitter off-springs. The Truncation Selection discards all but the best μ individuals. This is similar to the Roulette Selection, where individuals are selected in proportion to their fitness. A problem of these two selection variants is that, they tend to converge to a random selection if the fitness values are close. This problem is avoided in the Tournament selection [109], which select the fittest individual of some

[1] A Java-based Evolutionary Computation Research System - http://cs.gmu.edu/ eclab/projects/ecj/

t individuals picked randomly.

"Tournament Selection"

```
P := population
t := tournament size

best := individual picked at random from P
for i from 2 to t do
  next := individual picked at random from P
  if Fitness(next) > Fitness(best) then
   best := next

return best
```

More exploitative (i.e., parents stay longer in the population and compete with their children) selection algorithms like Elitism directly inject some fittest individuals into the next generation [109]. Other variants are steady-state approaches where successively a small part of the population is substituted [110] or a tree-style genetic programming approach. The first one iteratively breeds one or two children, accesses their fitness and reintroduces them into the population by killing some existing individuals. The second one is a special breeding technique developed by Koza [98] where a non-terminals of the individual are selected with a certain probability and a crossover is performed. These methods are often combined with single-state methods like hill-climbing, simulated annealing or iterated local search in hybrid selection methods. Therefore flexible GP pipelines are key to an easy configuration. All variants support probabilities to select terminals, non-terminals and root nodes.

Mutation

GP mutation picks a node or edge with a certain probability and tries to add, delete or change a node or edge. If the GP is strongly typed only type matching nodes should mutate (see section 4.2). The selection of nodes is handled like in 4.1.2. Mutation can mutate subtrees with randomly-generated subtrees, swap with (type compliant) subtrees and put constraints mutation like no subtree to be introduced should be contained in the existing one. Ephemeral random constants are mutated with some noise - often in relation to additional parameters like fitness.

Crossover

GP crossover swaps sub-trees and nodes with a certain probability. In GP crossover is a subset of mutation. Other representations (e.g., bit vectors in Genetic Algorithms)

often distinguish crossovers by the number of crossover points. A one point crossover exchanges all subnodes of the identified crossover point. A two point crossover limits the exchange to an identified sub-node of the crossover point. A third variant, the so called "Uniform Crossover" crosses a number of single nodes (without sub-trees) of the two individuals. In general crossover - as mutation subset - combines (only) existing nodes but does not produce new nodes.

4.1.3 Automatic Defined Functions

Automatic Defined Functions (ADF [108]) exploit the symmetry of GP solutions by mapping sub-trees to an ADF, which can be used like a normal node. The modularity exploits the heuristic that good (partial) solutions tend to be repetitive. ADFs were introduced and discussed in great detail in [108]. ADFs can speed up the convergence of the GP algorithm in two ways. The first exploits symmetries in the evaluation of GP nodes. The second one targets the modularity of the solution and their higher probability to describe the solution compared to GP types not using ADFs as a kind of macro context.

4.1.4 Fitness Computation

The fitness computation determines the quality of the individual and depends highly on the application context, see section 5.4, page 163. Nevertheless there exist different fitness representations for different problem categories, which are described shortly in this section.

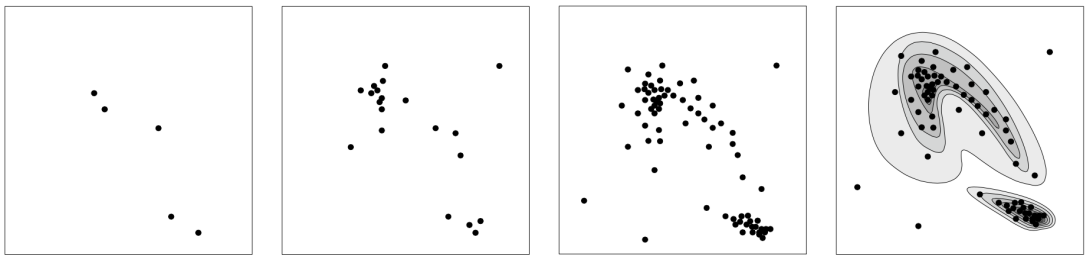
Co-evolution refers to the aspect that the fitness of individuals is mutually dependent, in a positive (cooperative co-evolution, e.g., different soccer players in a team, whereby each team member is represented as an individual) or negative way (competitive co-evolution, e.g., two soccer teams, whereby each team is represented as an individual). In contrast to the previous approaches there exist not an absolute fitness but a relative one. But this reveals a (complexity) problem for example regarding the question how the (relative) fitness is computed used in the breeding step. This problem is solved by test computations against the relative fitness, thus the fitness of one member of population P is evaluated against some members k instead of all members $\frac{|P|\times|P|-1}{2}$.

Multi-objective optimization tries to identify a “Pareto Front” in the space of candidate solutions of a fitness function set. The Pareto Front is the area of solutions where no fitness function dominates another fitness function. The “Pareto Front Rank” defines a qualitative value how far the individual is from the Pareto Front. Individuals of Rank 1 are the Pareto Front. Individuals of Rank 2 are the new Pareto Front that is computed after all individuals of Rank 1 are removed. The Pareto Front Rank is

equal to the fitness of the individual. Additionally "sparsity" is used to distribute the individuals of a Pareto Front Rank equidistant across the Front. Another idea of fitness computation is to use the individuals "weakness". The weakness of an individuals is proportional related to the number of individuals that dominate the individual. An algorithm implementing the Pareto Front members around this "weakness" concept is the "Strength Pareto Evolutionary Algorithm" (SPEA) [111].

Combinatorial optimization tries to solve problems like the "Knapsack" or "Traveling salesman" problem, where the combination of several unique components forms the problem solution. The problem solution has additionally hard constraints (e.g., in the knapsack problem the maximal height). This hard constraints should be taken into consideration in early stages of the GP algorithm. There exist different approaches to reward (Ant Colony Optimization) or punish components (Guided Local Search) that had a (presumable) influence on keeping the constraints.

Model Fitting tries to generate a model to find new individuals. Model Fitting by Classification tries to solve a binary classification problem by dividing the population into "fitter" and "unfitter" regions. There exist several of binary classification algorithms in the machine learning world like Decision Trees, Support Vector Machines, k-Nearest-Neighbor. Normally these classification algorithms produce models that can describe but not generate new individuals. A possibility to create algorithms that are able to generate new individuals are algorithms using a mathematical function to describe the distribution of an infinite-sized population called "Estimation of Distribution Algorithms" (EDAs). Figure 4.4 visualizes the idea exemplarily of population candi-



(a) A population of 5 individuals

(b) A population of 20 individuals

(c) A population of 75 indivdiuals

(d) A distribution of an infinite number of individuals, with Subfigure (c) overlaid for reference.

Figure 4.4: Distribution of population candidate solutions using samples 5, 20, 75

date solutions, using samples 5, 20, and 75 plus and infinite distribution (compare [93], p. 158). There exist different representations to describe the population. Examples are shown in figure 4.5 approximating the distribution of figure 4.4 with a histogram

three multivariate Gaussian curves and a marginalized version (compare [93], p.159 and p.160). The different approaches mainly influence the computational effort that

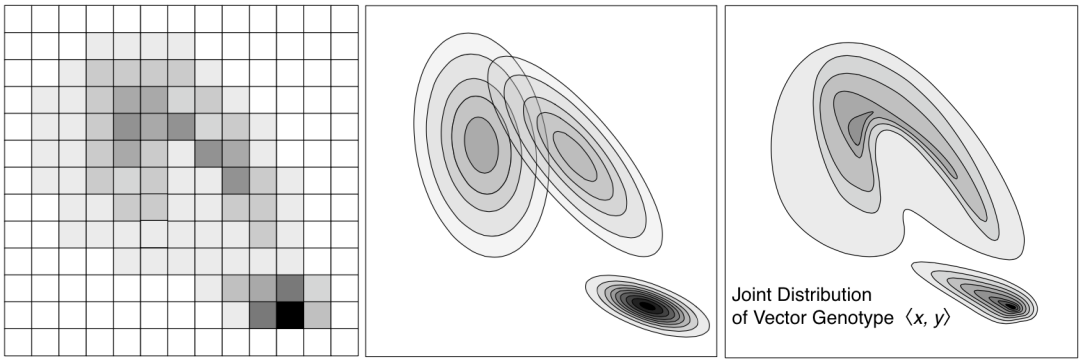


Figure 4.5: Approximating the distribution of candidate solutions with a histogram three multivariate Gaussian curves and a marginalized version

is required to compute the representations and (of course) the precision of the representation towards the real population. There exist several EDA variants for different domains, e.g., Population-Based Incremental Learning [112], Uni-variate Marginal Distribution Algorithm [113], or the Compact Genetic Algorithm [114].

As mentioned in the introduction the evaluation targets a hybrid approach. The evaluation is enumerative, which is possible because "real world data" is used to compute the fitness and restricting possible interactions to interactions tracked. The implementation uses a lightweight Model Fitting implementation to distinguish "fitter" from "unfitter" model parts. This information is used to optimize the GP algorithm.

4.1.5 Concurrent Computing

There are different ways to speed up Genetic Programming with concurrent computations. An obvious parallel granularity is an individual or a set of individuals that can be bred or evaluated concurrently. The computation can be done in multiple threads on multiple systems. The possible speedup with concurrent computation below this level depends - obviously - on the algorithm to compute the fitness and to breed new individuals. This is often application specific and not discussed in this section.

When the fitness evaluation is done by a set of machines it is called “Master-Slave Fitness Assessment”. When the breeding takes place on multiple machines it is called “Island Model”. Each island sends and receives a subset of the complete population and computes their fitness. This can also be combined, where an Island Model breeds

the individual and each island distributes the fitness computation in a "Master-Slave" configuration. Usually the best individuals are shared between the islands. Therefore one interesting aspect is the question of the optimal exchange ratio between a set of machines implementing the Island Model. Skolicki [115] identified fitness functions in his Phd thesis [116] suggesting that there exists an interval size of interchanged individuals increasing the solving capabilities of the GP algorithm. This effect is presented in figure 4.6. The Island Model is the only kind of parallelism that distributes the

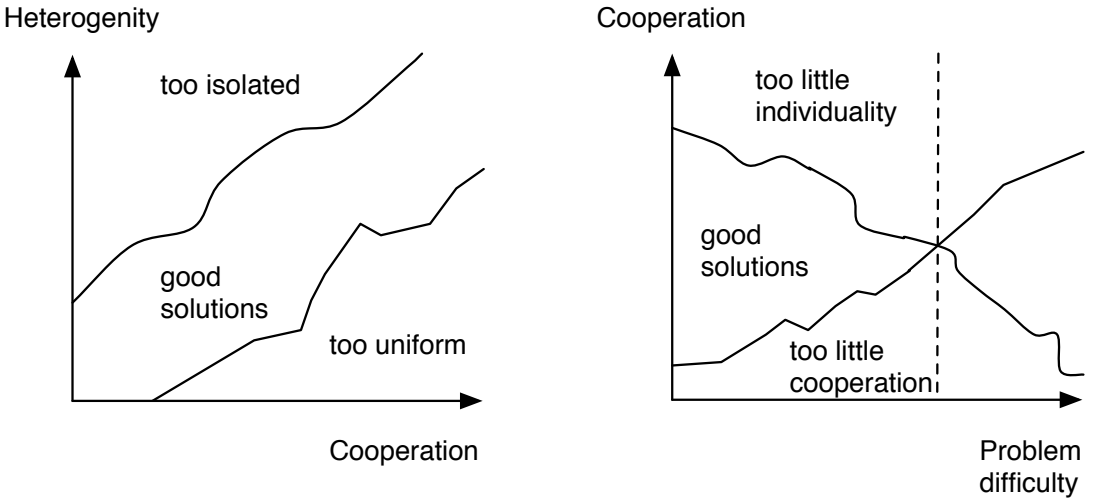


Figure 4.6: Hypothetical relations between cooperation and heterogeneity level and heterogeneity and solving capability

population. Another concurrent computation variant is the asynchronous computation, i.e., instead of waiting that the computation of all nodes has been done like in a Master-Slave scenario, the computation continues directly when nodes return their results. Another special variant is a so called "Spatially Embedded Model", where the physical location of the individual is embedded in the population and individuals are only allowed to breed with "nearby" individuals, so a good individual cannot spread as fast in a population as it could without this breeding constraint. The next section discusses the relationship between RMOF and Genetic Computation.

4.2 Genetic Programming and RMOF

The RMOF environment supports the generation of strongly typed parameter files for the ECJ algorithm. For this purpose a graph rewriting approach is used. First the layer file is loaded into the editing environment. Then the classes and objects are specified to be matched (left-hand site or in the EBNF LeftHandSite). Then the classes to set

(fix) or to generate are specified. Figure 4.7 shows an RMOF screenshot. Number 1 in

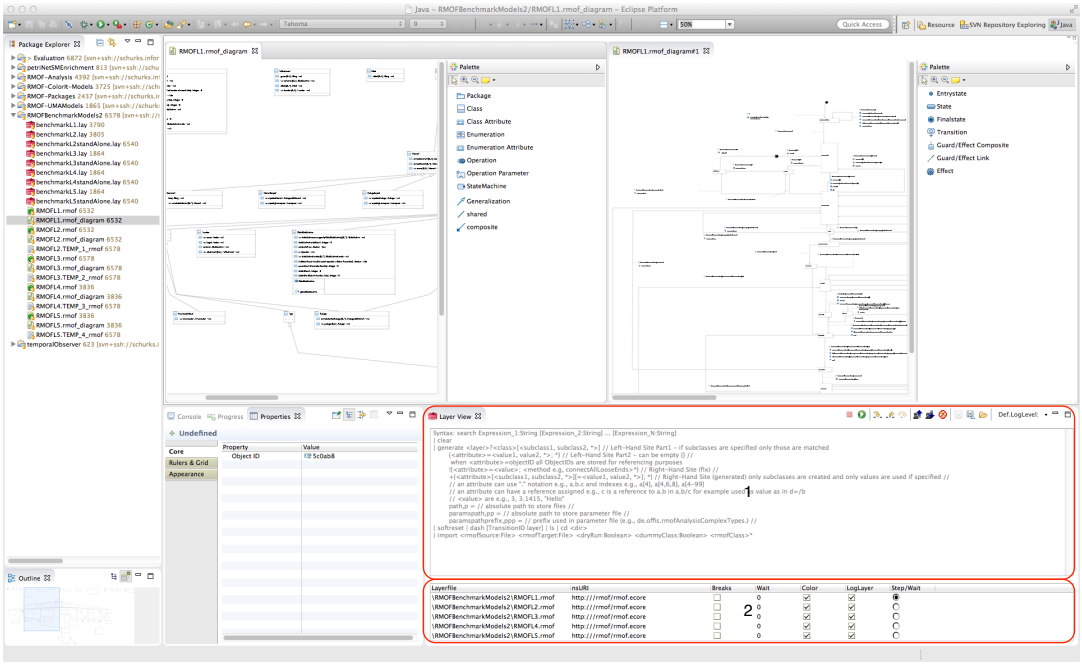


Figure 4.7: RMOF Screen with loaded models and RMOF editor

the screenshot presents the RMOF editor to specify the generation rules and number 2 the layer composition view. The editor supports the following syntax to generate the ECJ parameter files.

```
DigitNotZero      ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
Digit             ::= '0' | DigitNotZero;
NatNum            ::= Digit | (DigitNonZero { Digit } );
RealNum           ::= Digit { Digit } '.' { Digit }
Letter            ::= 'a' | ... | 'Z'
String            ::= Letter { Letter | Digit }
Class             ::= String { '::' String }
Reference         ::= String
ObjectInstances   ::= NatNum | { NatNum } | NatNum '-' NatNum
Attribute         ::= Class '.' String { '.' String }
                      [ ObjectInstance ]
                      [ '/' Reference]
Value             ::= '"' String '"' | NatNum | RealNum
```

```
Values           ::= '<' Value { ',' Value} '>'
AttributesValues ::= '{' Attribute '=' Values
                     { Attribute '=' Values } '}'
AssignPart       ::= '!' AttributeValue
GeneratePart     ::= '+' Attribute ClassesOpt [ Values ]
ClassesOpt       ::= '[' {Class} ']'
Layer            ::= NatNum
LeftHandSite     ::= Layer '?' Class ClassesOpt AttributeValue
RightHandSite    ::= AssignPart GeneratePart
GenerateCommand  ::= 'generate' LeftHandSite RightHandSite
                     { LeftHandSite RightHandSite }
```

The LeftHandSite of the EBNF matches the classes and attribute for the assignments (fix values) and generations of the RightHandSite. RMOF makes sure that the types of the attributes match and generates a strongly typed parameter file for ECJ. Strongly typed genetic programming [117] assigns type constraints and differentiates between so called atomic and set types. A value of an atomic type is a symbol or integer. A value of a set type is a set of atomic types. RMOF generates the type hierarchy according to the models loaded and the generation command specified. The following is an extraction of a generated parameter file.

```
# atomic types subpopulation 0
gp.type.a.0.name = NIL_0
gp.type.a.1.name = ROOT_0
gp.type.a.2.name = InteractionPattern_0
gp.type.a.3.name = LeftButtonMouseInteraction_0
...

# set types subpopulation 0
gp.type.s.0.name = Interaction_Set_0
gp.type.s.0.size = 6
gp.type.s.0.member.0 = LeftButtonMouseInteraction_0
gp.type.s.0.member.1 = MiddleButtonMouseInteraction_0
...

# constraint Interaction_Container_0
gp.nc.0 = ec.gp.GPNodeConstraints
gp.nc.0.name = ncInteraction_Container_0
gp.nc.0.returns = Interaction_Container_0
gp.nc.0.size = 2
gp.nc.0.child.0 = Interaction_Set_0
```

```
gp.nc.0.child.1 = Interaction_Container_Set_0

# Instances Function Sets
gp.fs.0.func.0 = ec.rmof.analysis.Instance
gp.fs.0.func.0.nc = ncInteractionPattern_0
gp.fs.0.func.0 = ec.rmof.analysis.Instance
gp.fs.0.func.0.nc = ncLeftButtonMouseInteraction_0
```

The extraction of the generated parameter file starts with some atomic types. Then a set types with six members is presented. Two members are specified. The set type `Interaction_Set_0` can be instantiated with a `LeftButtonMouseInteraction_0` or `MiddleButtonMouseInteraction_0`. The third block describes the constraint `Interaction_Container_0` that specifies the number of children. In this case `Interaction_Container_0` has two children an `Interaction_Set_0` and an `Interaction_Container_Set_0`. After the constraints have been specified they are mapped to functions that need to be implemented. In this case the constraints `ncInteractionPattern_0` and `ncLeftButtonMouseInteraction_0` are mapped onto the class `ec.rmof.analysis.Instance`.

The previous extraction of a generated parameter file shows the mapping between RMOF and the ECJ framework. Besides these parameters there exist others to specify the concurrency level (see 4.1.5), checkpointing, statistics, breeding pipelines and various RMOF dependent problem parameters (e.g., path of the tracking database).

An analysis run uses the RMOF environment in the following way.

(i) (The existing RMOF layer models are loaded initially.)

(ii) The GP algorithm computes GP instances of the models elements to be generated.

(iii) The GP instances are mapped onto RMOF models. The object cache stores all model changes and results before they are actually written into the corresponding EMF model. The cache differentiates two kinds of resets. The first one, a hard reset, switches back to a defined state where no model change took place. The other one, a soft reset, stores (with the Least Recently Used (LRU) tactic) all newly created models to speed up the creation process if possible.

(iv) The simulation source code of all RMOF layers is generated, compiled, and executed.

(v) Fitness values (and additional feedback information) is extracted from the RMOF models or the environment. The environment gathered the information from all evaluation instances. In a Master-Slave configuration this is constantly transferred over the network (e.g., cache statistics and feedback information). The fitness information is directly encoded in model elements on all RMOF layers.

(vi) RMOF models are reset to the state of step (i).

(vii) (Continue with step (ii) until the fitness is high enough.)

(viii) If the fitness value is high enough or the maximum number of generations has been processed a last model is evaluated and additional model informations are introduced (e.g., computation of coverage values) and stored into a specified model file of the ECJ parameters.

The next section discusses GP optimizations, which are often RMOF related.

4.3 Genetic Programming Optimization

The GP optimization described in this section discusses optimization techniques that are all RMOF related but model independent.

4.3.1 Data Access and Precomputations

The tracking database is technically an API providing access to all relevant tracking informations. The tracking information is stored in XML files. An online parsing during an analysis run would take too much time. The tracking database stores all information in a compact form that can be used fast and with low memory consumption. Additionally several precomputations are done e.g., to find valid GUI abstractions. Which information is relevant can be configured in parameter files. The computation of the tracking database is done concurrently and also includes a detection and reduction of temporal gaps. A temporal gap is the time between interactions considered to be too long for a non activity. The following log extract shows some parsing logs without and with a time gap.

```
...
20. Parsing output_xml file "/Users/lobe/Trackings/12080809123
    +0200/trace120808091236795+0200.output_xml.annotated"
19. Time (tracked - no gap detected) : 1m:12s:332ms
21. Parsing output_xml file "/Users/lobe/Trackings/12080810521
    +0200/trace120808105210459+0200.output_xml.annotated"
7. Time (tracked - no gap detected) : 5m:34s:128ms
```

```
22. Parsing output_xml file "/Users/lobe/Trackings/12080909132
    +0200/trace120809091323676+0200.output_xml.annotated"
18. Time (tracked - no gap detected) : 1m:44s:773ms
23. Parsing output_xml file "/Users/lobe/Trackings/12081012530
    +0200/trace120810125302370+0200.output_xml.annotated"
23. Time gap detected of length: 2d:19h:39m:11s:736ms,
    reduced to: 10m:0ms
23. Time (tracked - with gaps) : 2d:19h:41m:19s:839ms -
    (detected gaps) : 2d:19h:29m:11s:736ms
    = (tracked - gaps removed) : 12m:8s:103ms
24. Parsing output_xml file "/Users/lobe/Trackings/12081308443
    +0200/trace120813084437939+0200.output_xml.annotated"
2. Time (tracked - no gap detected) : 7m:50s:191ms
...
```

After the initial parsing the tracking database creation computes valid, unique values and abstractions of all GUI elements. GUI elements refer a complete path of composed GUI parent element to identify the GUI element. This is required because Eclipse assigns new IDs to all GUI elements after a restart of the environment. These GUI paths include the concrete position of the child GUI elements in the parent. The GUI abstractions abstract from these position information and aggregate effectively GUI classes. The GP algorithm chooses between abstractions and unique GUI IDs. The following log extract shows some computation logs entries during the computation of abstractions of some few tracking files.

```
Find abstract Mouse Widgets
0.000% done. Interactions required to be an abstraction: 10
0.5882% - Time required = 2ms - Time TBD = 338ms
        - abstractions found : 0 - coverage : 0.0000%
3.5294% - Time required = 5ms - Time TBD = 136ms
        - abstractions found : 1 - coverage : 1.9286%
...
98.2353% - Time required = 1s:322ms - Time TBD = 23ms
 - abstractions found : 22 - coverage : 95.9988%
99.4118% - Time required = 1s:323ms - Time TBD = 7ms
         - abstractions found : 22 - coverage : 95.9988%
100%.

Abstract widget IDs  :
7 (Keyboard, unique: 181, coverage: 99.3225 %),
22 (Mouse, unique: 170, coverage: 95.9988 %)
```

```
= 29 (Total, coverage: 97.66065 % (mean))
Unique/abstract IDs (bef.merge) : 351 + 29 = 380
```

The main purpose of the tracking database besides the temporal corrections and the computation of abstractions is the fast access to the data itself. There adequate data structures with a low memory consumption are used to map the relevant data of all tracking files. No need to say that the database itself is thread-safe.

4.3.2 Caches

Caching is done on several layers in RMOF. There is a model element caching of introduced GP elements. Once a GP element has been introduced to RMOF on one computation node it's reused if it's still in the cache and not recreated. Fitness computation caching is done during the behavioral pattern recognition analysis. The fitness of a sequence correspond to the number of successful mappings on the tracked data. The 1st level sequences cache caches the sequence fitness of independent interaction sequences. The independent fitness points are used to filter sequences with zero fitness points. The 2nd level sequences caches the sequence points of the merged interaction sequences. All cache sizes can be configured in the GP parameter files and are based on LRU. The following log extract shows additional efficiency stats of the caches used:

```
Thread : 42, '42' = 46.54%, '58' = 4.41%, '74' = 0.98%,
         '90' = 0.73%
Thread : 43, '43' = 39.95%, '91' = 5.85%, '59' = 4.83%,
         '75' = 1.53%, '107' = 0.51%
Thread : 40, '40' = 42.08%, '56' = 4.28%, '88' = 4.03%,
         '72' = 1.76%, '104' = 0.50%
...
```

These values are cumulated from all evaluating threads respectively nodes (in Master-Slave configurations) in the GP run statistics. The next log entries show some overall cache stats.

```
1st Level Cache Hit Rate=53.79% {27.30%} (27.30%) [27.30%]
2nd Level Cache Hit Rate=52.67% {27.00%} (27.00%) [27.00%]
```

The first entries are the cache hit rates of the current GP generation, the next of the last 10, 100, and all previous computed GP generations.

4.3.3 Feedback Loops

The feedback loops influence the probabilities of the GP algorithm to select, brew, mutate, and crossover (if parameterized). If a sub-model respectively subset of GP nodes

of an individual has no positive fitness it is changed with a high probability. Characteristica of model parts that are changed with a high probability are successively filtered and chosen with a low probability resulting in a light weight Model Fitting approach (see 4.1.4, page 146).

The next chapter introduces the evaluation of RMOF and the analysis and synthesis methods and techniques developed in this thesis.

5 Evaluation

This chapter describes the evaluation of RMOF and the implemented methods and techniques to synthesize, analyze and optimize abstractions of development processes. The evaluation goal was to apply the developed methods and techniques in an industrial context including a relevant size of models and interactions. This context can be easily changed to different scenarios to optimize different development activities like requirements engineering, design, or implementation. For all these scenarios it is important to understand the limitations of the developed approach. These limitations are part of the evaluation not the results or the result validation including in particular the evaluation of the suggested optimizations by changing the test tool. This should be topic of additional controlled experiments where these methods and techniques are applied. Nevertheless the validation of the method itself is relevant because of the meta-heuristic components of the approach and is discussed at the end of this chapter.

A development process and its activities and artifacts to be optimized were automatically tracked by software components developed in this work (see 5.1) and was introduced into the testing activities and environment of an industrial partner. Interaction patterns were synthesized on the tracked data. The patterns were embedded into a process model, analyzed and the sequences were optimized with respect to their impacts on the interaction effects (an interaction effect being for example a manual and successful test vector change of a software in the loop test activity). Model characteristics and semantics with an impact on the interaction patterns were identified to prognose testing efforts based on model characteristics developed in previous development steps.

The chapter starts by introducing the industrial partner and the software that was enriched to support the automatic data acquisition. Section 5.1 presents the software that was developed to track the data automatically. Section 5.2 briefly sketches correction and enhancement of the tracking files, e.g., to detect abstract interactions (like copy and paste of test vectors) or the detection of test successes and failures for each test execution. Section 5.3 presents an abstraction of Matlab Simulink Stateflow models used to meet confidentiality agreements with the industrial partner. Section 5.4 introduces the interaction pattern detection algorithm that was used and the resulting interaction sequences found. The found interaction sequences are investigated in section 5.5 and some (long) sequences with a high impact are optimized. Impacts are

computed by simulating the interaction sequences, which is topic of section 5.6 before section 5.7 concludes by introducing properties (e.g., a kind of complexity idea of input models) to guide the process models discovered further.

The data collection took place at BTC Embedded Systems AG[1] between 03/2012 and 02/2013. BTC Embedded Systems AG with offices in Oldenburg, Munich, Berlin and Tokyo is part of BTC Business Technology Consulting AG – one of the top 20 IT software companies in Germany. They have been successfully involved in the automotive industry on an international level for more than ten years, and have been instrumental in the success of well-known manufacturers and suppliers. BTC Embedded Systems AG developed several products like BTC EmbeddedSpecifier, BTC EmbeddedValidator, BTC EmbeddedTester and BTC Testvector Editor. BTC Testvector Editor (TVE) is an Eclipse based environment used to specify test vectors and test Matlab Simulink models. Model in the Loop testing triggers a Matlab Simulink Simulation. Software in the Loop testing can use DSpace TargetLink as code generator. The results of a test execution are presented in the environment after the test execution took place. TVE supports several XML based test vector formats, including cell highlighting and special views to filter signals or display value changes over time. Figure 5.1 shows a TVE screenshot. Number 1 in the screenshot represents an editor for the test vectors. The selection of test steps and the simulation mode is done in view number 5. The results of the test execution are presented in additional TVE editors (number 2 in the screenshot) with used-defined colors to differentiate successful and failed test cases. The additional view 3 show test signal propagation, 4 test vector filtering, and 6/7 test specifications. Parts of the screenshot are intentionally blurred because of intellectual property reasons (e.g., signal names).

The tracking of the test activities was done in approx. 400 sessions. Each session is recorded in a XML tracking file. All tracking have a total volume of approx. 25 GB data. During the tracking 60 different Matlab model files have been tested. The model files are parsed and serialized in so called "Silnab" model files (see 5.3, page 162), an EMF based Matlab Simulink abstraction file format preserving intellectual property rights. Time gaps (pauses in the tracking of more than five minutes) were reduced to five minutes. The total tracking time was 62 days and 22 hours non-stop testing activities after reduction. This corresponds approx. to one person year[2] with an average session time of approx. 1,5 hours. The next section presents the software that was used to collect the data automatically.

[1] http://www.btc-es.de
[2] http://www.tagesspiegel.de/wirtschaft/arbeitszeit-1650-stunden-arbeit-pro-jahr/1834390.html

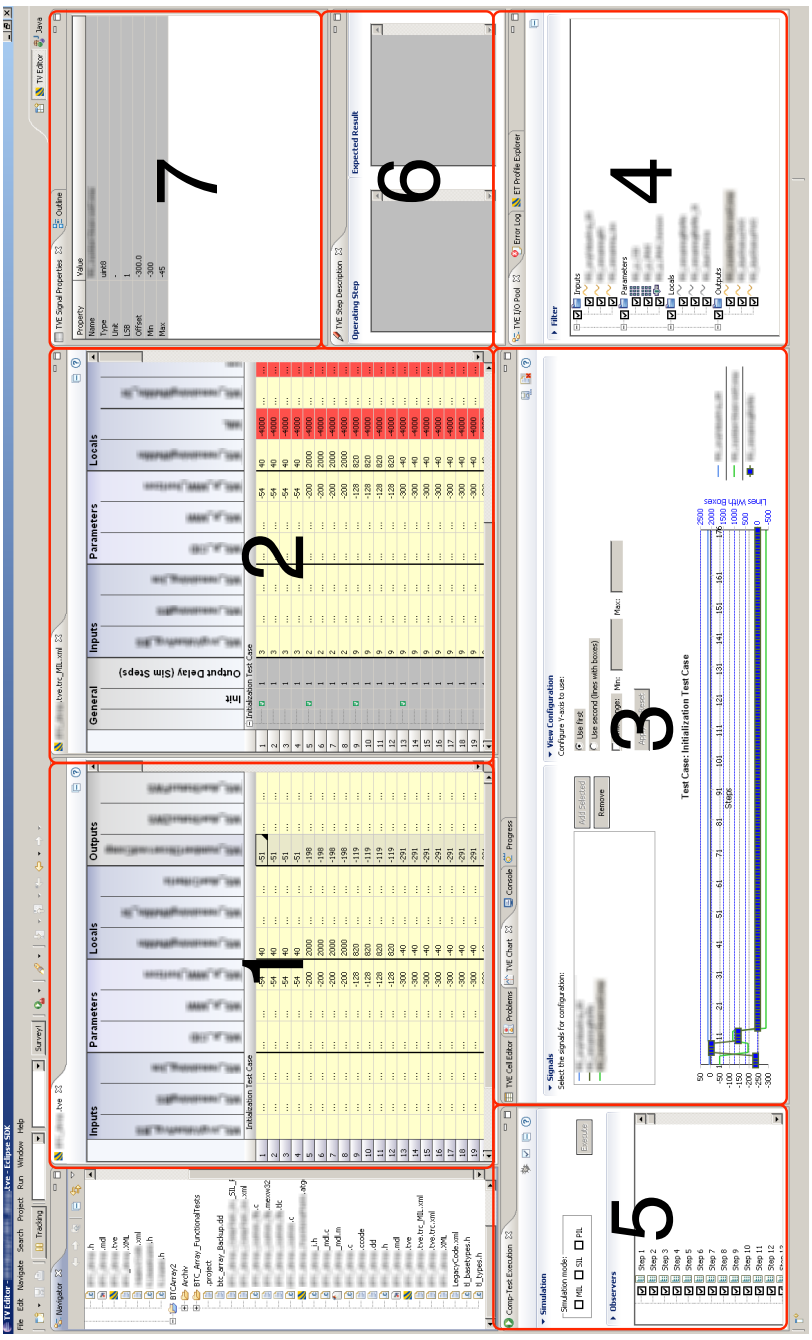


Figure 5.1: BTC Testvector Editor screenshot

5.1 Interaction Tracking

TVE is based on Eclipse and GUI elements of Eclipse base on the Standard Widget Toolkit (SWT [118]). The SWT is a native interface to the underlying graphical user interface of the platform e.g., Windows, Cocoa, or Motif. SWT GUIs provide maximal performance and native application look and feel.

SWT has a special thread dealing with all kinds of GUI events called "SWTDisplayListenerThread". This thread can be extended with listeners for different kinds of SWT Events. With such an extension it is possible to capture mouse and keyboard events. The "event" class includes attributes like mouse button, relative X position, or state mask (e.g., when a control button is pressed during the interaction). The "event" also refers to the GUI widget that has been used during the interaction. The GUI elements need to be parsed to retrieve additional information. The GUI elements range from buttons, over menus to complete TVE test grids. A TVE grid is a kind of Excel sheet to edit test vectors and display test execution results (numbers 1 and 2 in figure 5.1). The SWT tracking software built to track user interactions is complemented by the Usage Data Project [119] to capture bundles (also known as plug-ins) that are started by the system, commands accessed via keyboard shortcuts, actions invoked via menus or toolbars, perspective changes, view and editor open, close, and activation events (activations occur when a view or editor is given focus).

Besides GUI interactions all Matlab files that are tested during the tracking are stored in Silnab models (see 5.3, page 162). The tracking software can be configured in its preferences. Figure 5.2 shows a screenshot of the tracking preferences.
The preferences allow the specification of the tracking directory, encryption and zipping of tracked files, or the setting of the log level. The log level influences the console output and acts as filter to the tracked information and was mainly used for debug purposes. The preferences allow the management of all tracking files to view, manually zip, encrypt, delete or clear the files from the log.

The tracking is simply started by pressing the tracking button in the Eclipse environment (number 1 in screenshot 1.5 on page 13). Additional (manual, arbitrary) information can be introduced into the logs by using the survey dialog elements.

The tracked data shows that approx. 2.000 views were opened and closed, approx. 500.000 mouse interactions and approx. 350.000 keyboard interactions took place, and precisely 3.364 test executions were detected. Besides these interaction trackings the changes in the TVE editing grid were continuously parsed and added in the logs e.g., with over 35 Mio. changes in TVE cells (changes due to test executions or TVE editing operations) with approx. 350.000 view changes of the TVE Grid (scrolling or re-sizing).

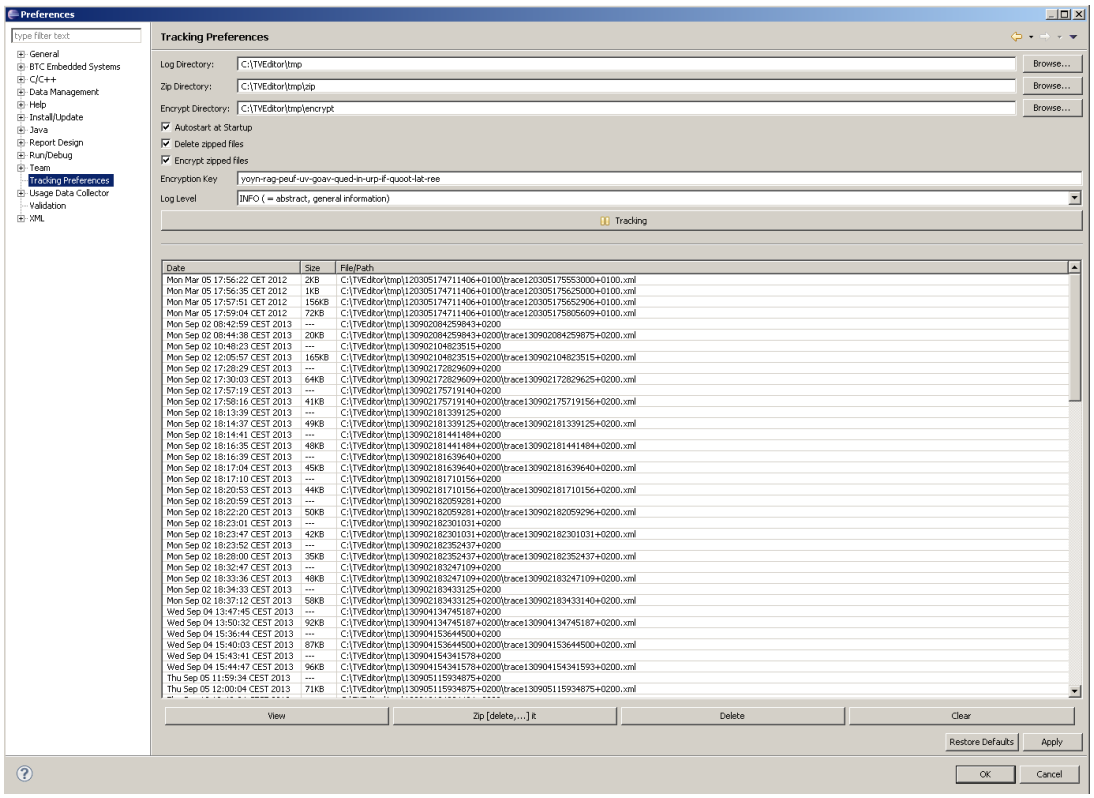


Figure 5.2: Process information tracking preferences

The content of the next section is the correction and first processing of the log files.

5.2 Corrections, Interpretations and Aggregations

First, the XML tracking files needed to be corrected because of malformed XML and incomplete parsed TVE grids. This can occur when the tracking software was shut down due to an exception. When an exception occurs the plugin disables itself and dumps a trace to the interaction log/console. The main target was not to interfere with the testing activities.

Second, TVE grid change blocks are numbered uniquely. A block is a consecutive number of changes in the log entries. This is done to identify results of TVE grid edit operations. For example a copy and paste operation or a test execution can change a set of test vectors.

Third, copy/cut, paste, and highlight operations are detected. For this purpose the TVE grid is reconstructed inline with the log entries. The log entries capture only the delta between the previous and the current TVE grid content. The detection also parses grid switches (switching from one TVE grid to another). There are manual grid switches and automatic ones e.g., when the test results are presented in an additional result TVE grid. The test results are parsed regarding compliance (successful test cases) and deviations (unsuccessful test cases) and differentiated into Model in the Loop (MIL), Software in the Loop (SIL), Processor in the Loop (PIL) tests. The editing operations that took place before each test execution are classified (e.g., manual, copy and paste changes or highlight operations) and assigned to the corresponding test execution. They are used as a filter when successful and unsuccessful tests are parsed.

Fourth, model files are associated to tracking files. The association is required to add missing model files for each tracking if no test execution took place. This is done by matching the content and time differences of the TVE grids edited with model files assigned and assigning the model files to TVE grids with no model file assigned. The Matlab Simulink files that are tested during a tracking session are abstracted and stored in Silnab Models. These models are described in the next section.

Fifth, interactions are related to model files. The association is required to connect (the influence of) model files to the interactions. During this step other information like currently open views is also added to the interactions.

5.3 Silnab Models

Silnab models are partially abstracted, scrambled and anonymized Matlab Simulink models. To create these models, the Matlab Simulink model used during a test execution is parsed and mapped to a Silnab EMF instance. Silnab models were implemented to meet confidentiality agreements of the involved industrial partners (e.g., omitting the engineers name). The scrambling clouds, e.g., the concrete block function and the abstraction is used to identify variables involved in comparisons. The abstraction allows a data flow reconstruction but hides the concrete function of the model as well as personal/company information. Figure 5.3 shows a screenshot of a Silnab editor. Number 1 shows a graphical (GMF based) editor presenting Matlab blocks and connections. Number 2 presents detailled information of a single Matlab Simulink block, e.g., Data-, Signal-type and Port. Number 3 shows the block-type in a property view. This information is scrambled with an appropriate hash function as shown in Number 4. Other information scrambled is e.g., the identification of the engineer who developed the model. The interaction tracking logs provide the base to compute interaction sequences. The detection of these sequences is discussed in the next section.

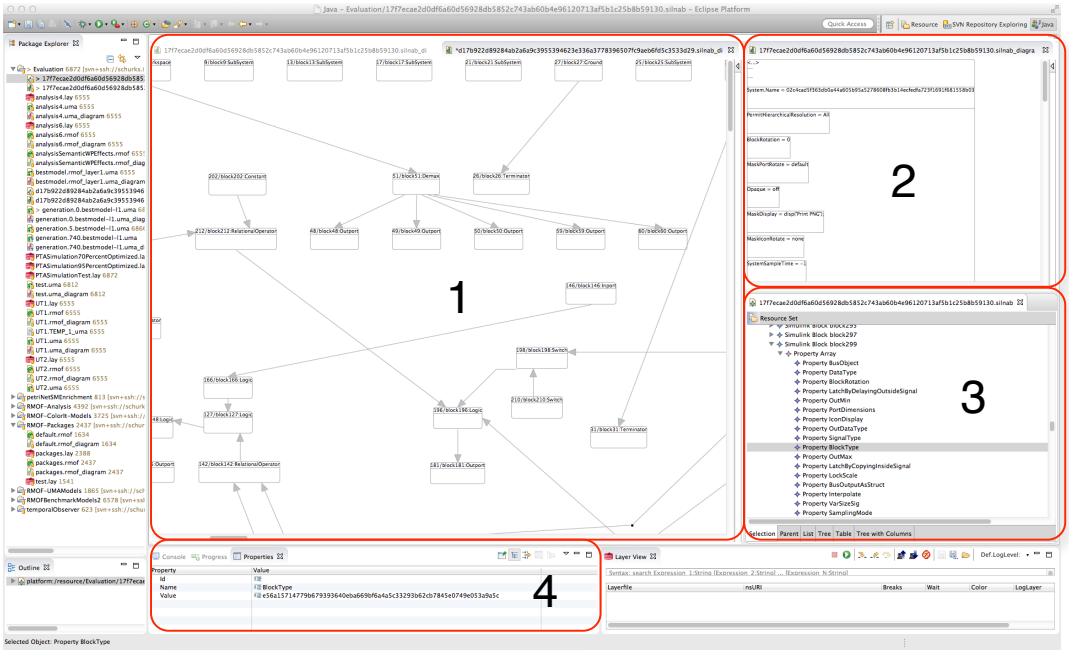


Figure 5.3: Screenshot Silnab editors

5.4 Interaction Pattern Detection

The pattern detection is done on base of Genetic Computation using a fitness function with a modified version of the Knuth-Morris-Pratt (KMP [120]) algorithm described in [121]. Before the KMP algorithm checks a pattern in a text it pre-computes prefix-sub-pattern of the pattern. If the pattern matching fails, it tries to match valid prefix-sub-pattern without the necessity to compare the previous text again - saving (some) comparisons. Figure 5.4 shows an example illustrating the idea of the KMP algorithm. Red indicates a failed comparison (at position i+1), blue the current position checked (where 'A', 'B' is not checked again). The KMP algorithm is $O(n)$ in the worst and average case for the searching phase. The pre-computation is also done with the KMP algorithm.

The algorithm is applicable for all kinds of patterns. It was extended to match interaction types (e.g., keyboard, mouse, wheel, and special void interactions = unknown interaction type) state-mask keys (e.g., CTRL pressed) and additional attributes (e.g., if a keyboard action takes place, what key was pressed) and in particular the (unique) GUI element where the interaction took place.

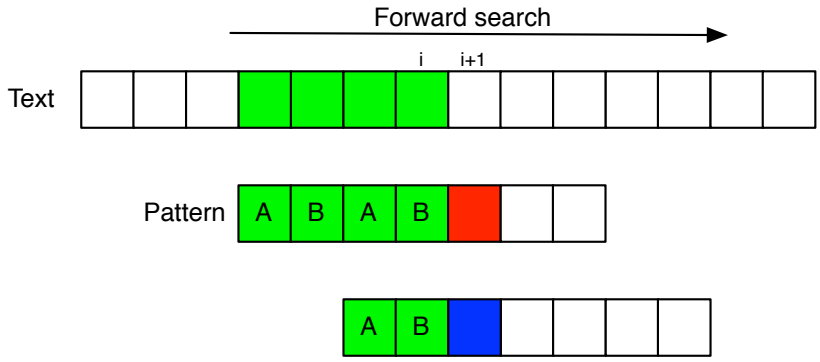


Figure 5.4: Idea of the KMP algorithm

The complexity of the algorithm is mainly driven by the possible permutations of the pattern length. Let's assume there are m unique interactions and l is the pattern length. The resulting permutation set has the size $s = m^l$. l should be less or equal m. If l (or m) is not limited, the problem is NP-complete. But if l is limited, the problem is P-complete, which is assumed to calculate some application numbers. The time required to identify sequences of a given length l can be computed by some profiling and application data. Every computation of the KMP algorithm takes approx. one second (which holds for patterns of size three). Let's assume there are 1.000 parallel nodes available and the possible parallel speedup scales. The tracking database shows 6.000 unique operations. The computation of all patterns of length two would require $6.000^2/1.000$ seconds = ten hours. A pattern of length three would require nearly eight years with 1.000 computing nodes. With genetic computations it was possible to find sequences of length nine and above with a coverage of 98% with some hundred computation nodes in less than a day.

The implemented fitness includes a disjunctive pre-evaluation phase of all patterns, a combined evaluation phase for all pattern, and the final computation phase of the winning sequences including the removal of not-winning interaction pattern in the complete mapped sequences. Fitness points are graduated (element matching points, disjunctive sequence matching points, joint sequence matching points) and weighted according to the sequence length. As already mentioned in 4.3, two caches were used to cache previous mapping results of single sequences and combined sequences. The following console snapshot shows the output of an example computation with 5 threads (evaluation and breeding) on a local host.

```
| RMOF-ECJ
| Evolutionary based analysis system for RMOF 1.41
| By Ralf Buschermoehle (RMOF), Sean Luke et al. (ECJ 21)
```

```
| URL: http://cs.gmu.edu/~eclab/projects/ecj/
| Mail: buschermoehle@demand2offer.de
| Date: August 22, 2014
| Current Java: 1.6.0_32 / OpenJDK 64-Bit Server VM-23.25-b01
| Required Minimum Java: 1.6


Threads:  breed/5 eval/5
Seed: 1699690226 1699690227 1699690228 1699690229 1699690230
Job: 0
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Deserializing tracking database (in thread 1) ... in 19s:784ms
Processing GP Tree Constraints
Setting RMOF Models Path: ~/Temp
Setting RMOF Layers File: ~/Temp/Evaluation/UT1.lay
Setting RMOF Parameter: 1?InteractionPattern{ia=objectID;
guiElements=objectIDsFromGUIIDs}+{interactions;
interactions.next; interactions.guiElement;
interactions.stateMask; interactions.asciikeys}
Setting Tracking DB: ~/Temp/trackingCache.ser
Initializing thread 1/5, ID 34 done.
Initializing thread 2/5, ID 35 done.
Initializing thread 3/5, ID 32 done.
Initializing thread 4/5, ID 33 done.
Initializing thread 5/5, ID 38 done.
Thread34,'1' = 1.76 %,'2' = 0.06 %,'3' = 0.02 %,'4' = 0.02 %
Thread35,'1' = 1.73 %,'2' = 0.04 %,'3' = 0.03 %,'4' = 0.02 %
Thread32,'1' = 1.62 %,'2' = 0.12 %,'3' = 0.05 %,'4' = 0.04 %
Thread33,'1' = 1.74 %,'2' = 0.05 %,'3' = 0.04 %,'4' = 0.02 %
Thread38,'1' = 1.81 %,'2' = 0.02 %,'3' = 0.01 %

Subpop 0 best Fitness: Standardized=629072.06 Adj.=1.5896404E-6
Hits=0 SeqCov=11.5469% ElemCovAllSeq=31.0432%
ElemCovSingleSeq=60.7606% ElemCov=60.9731%
LongestSL=2 MeanSL=1.2143 MedianSL=1.0000
s[1]=11,SC=10.8703%|74184,EC=55.7212%
s[2]=3,SC=0.6766%|4617,EC=5.2520%
GPSize=47, Individuals evaluated (gen/all): 4096/4096,
```

```
1st Level Cache =2.07%, 2nd Level Cache =1.85%
Individual successfully written to file "~/Temp/g.0.bI.ind".
Model written to file "~/Temp/g.0.bm-l1.uma", obj 238
Breeding: 828ms, Evaluation: 4m 08s 133ms (/I: 60ms),
Time passed : 4m 08s 961ms (/I: 30ms),
TTF (all): 14d 09h 42m 31s 40ms [14d 09h 46m 40s 1ms],
TTF (last): 28d 19h 25m 12s 78ms [28d 19h 29m 21s 39ms]
```

Each thread uses a seed to configure the random number generator. The seed can be set to 'time' (=random initialization) or to a fix number (e.g., to recompute a previous run). This is followed by parameter files processing (e.g., parsing attributes to generate GP nodes "Setting RMOF Parameter"), setup operations (e.g., reading the tracking database) concluded by the thread inititalization. Then the fitness evaluation starts and ends with cache statistics followed up by a short description of the best individual of the population (including e.g., sequential coverage of the different sequences and their length). Finally, an overall cache efficiency, serialization information and timings are written to the log. Each evaluation breeds the GP nodes, introduces the nodes as RMOF instances, triggers an RMOF simulation evaluating the fitness, reads and sets the fitness of the GP nodes and handles GP feedbacks, e.g., to mutate some nodes with a higher probability and/or larger "steps".

The computation was mainly done with a master/slave configuration. In contrast to a single computation node the thread sends the individual over the network to a slave, the slave evaluates the individual and returns fitness, individual and additional information related to caching and optimization back to the master node. The master node scheduled max. 500 computation slaves. Each slave normally controlled a single CPU, whereby some nodes controlled a set of computation slaves. The CPU cores used include AMD Opterons, Intel Xeons and Intel i7s with large performance differences. A fine granular scheduling mechanism on the master was required to use the nodes optimally. The master node was a 16 core Xeon distributing jobs and collecting the results from the slave nodes concurrently. Several modifications were introduced into the ECJ framework to optimize the computations including a flexible rescheduling mechanism if computation nodes are introduced and removed arbitrarily. This was necessary because the supercomputer nodes used predefined job wall times. Figure 5.5 shows two pictures of the High Performance Cluster 'Hero' in Oldenburg [1] where computations took place. The initial computations in 2013 required months (prognosis) to find patterns with a interaction coverage of $> 90\%$. After introducing several optimizations the computation times were reduced to six hours on 300 cores with a coverage of $> 90\%$.

[1] http://www.uni-oldenburg.de/fk5/wissenschaftliches-rechnen/hpc-facilities/hero/

Figure 5.5: High Performance Cluster “Hero” in Oldenburg

A final interaction pattern model often tends to be complex in its completeness and in its graphical dimensions. It consists of several hundred different interactions (on different GUI elements) and some thousand interaction connections between them. Figure 5.6 shows a complete interaction pattern model with 95% coverage. However, the representation makes it difficult to identify useful information. The RMOF environment can filter interactions and transitions in relation to their probability, compute abstractions and extract single sequences to visualize them appropriately. Figure 5.7 shows a single interaction sequence. The start interaction (black filled circle with a rectangle) is not connected, this means that the interaction sequence was not triggered initially in a session. Blue interaction transitions connect conditional interactions with a probability <= 100% (all). Black interaction transitions connect only unconditional interactions with a probability of exactly 100%. Two kinds of interactions are found in the model of figure 5.7, a left mouse button interaction and mouse wheel interaction. There are four interaction effects shown in the model. First, a scrolling depicted by the cross icon. Second, a scrolling in the TVE Grid visualized by an icon with a cross over some boxes. Third, a resize in the TVE grid presented by magnifier glasses. Fourth, a menu invocation presented as the blue icon. The interaction effect value describes a single effect (e.g., scrolling effect - mean pixels scrolled of all mapped interaction effect). In figure 5.7 all interactions refer to the same GUI element (connect with a dashed line). The connected GUI element is identified by the GUI path. The referred

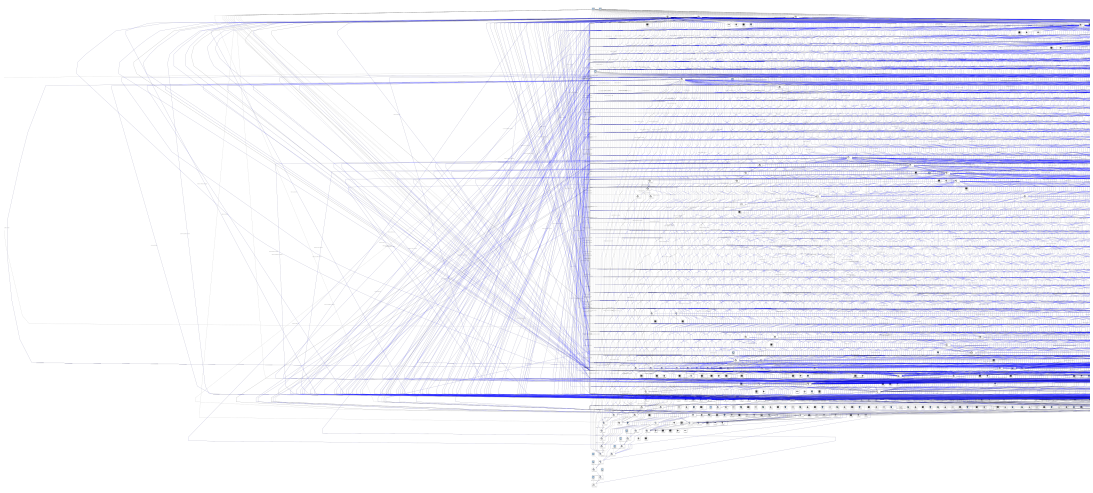


Figure 5.6: Partial interaction pattern model with 95% coverage

GUI element can be identified with another RMOF plugin that highlights GUI elements of a given GUI tracking path when the mouse cursor hovers a subpath. A screenshot of the GUI element identification tool is shown in figure 5.8 with a highlighted TVE grid in red. The GUI element of the interaction sequence points to a TVEGrid where nearly 30% of the interactions took place.

The patterns identified covered all interactions done with 96.6823%. The computation required roughly 17 hours with 300 cores. Table 5.1 shows the number of sequences, their lengths and and their sequence probabilities that were found:

The element probabilities are computed for each element of the sequence individually, neglecting that there can only be one sequence winning. This is represented in the sequence probability. There were approx. 4.3 Mio. individuals evaluated. The cache hit rate of the disjunctive sequences was 5%, and the combined sequences cache was 75%. The target sequence length was 9 (the fitness evaluation should evaluate sequences of length $9 * 2 - 1 = 17$ better then separate sequences). Figure 5.9 shows an overview on single interaction sequences with a coverage $> 0.5\%$. The sequences are ordered in clusters of the captured sequence length. In particular long sequences are interesting and sequences with a high temporal impact. These sequences offer simple and high optimization potentials. These optimizations are discussed in the following sub-sections.

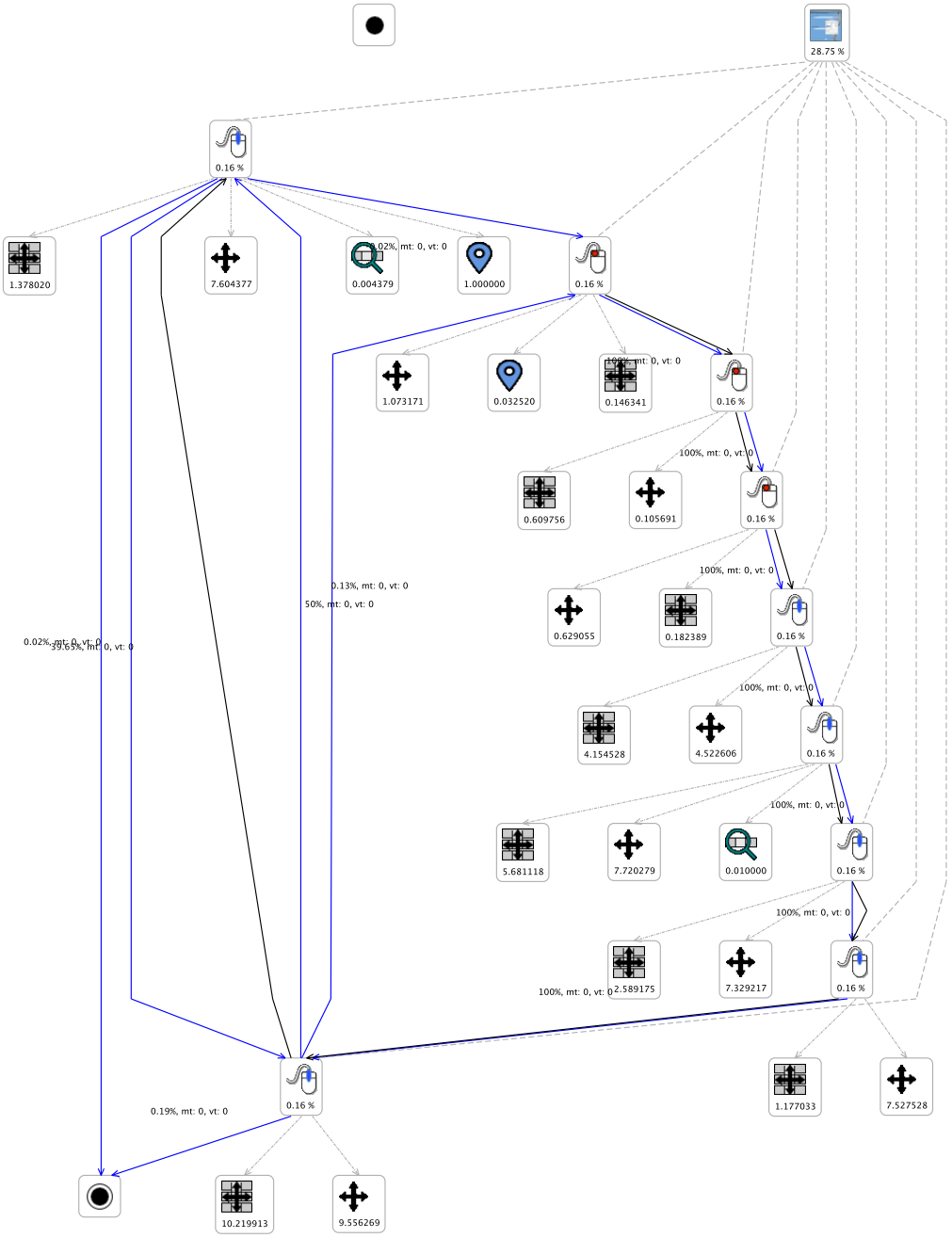


Figure 5.7: Single interaction sequence

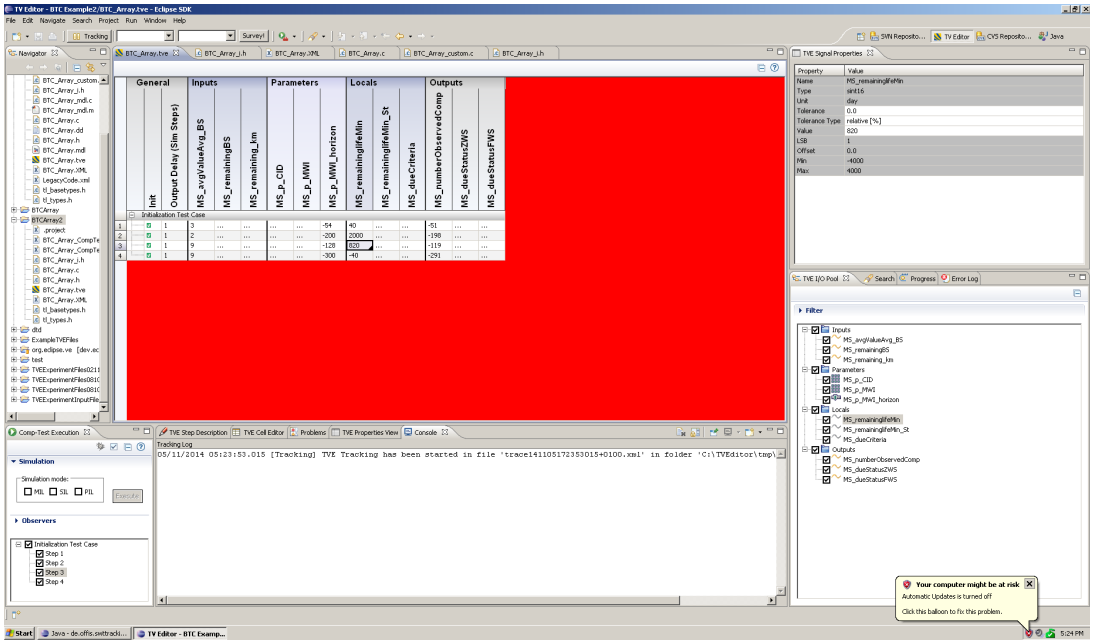


Figure 5.8: GUI element identification

Length	Occurrences	Sequence probability	Element probability
1	179	34.7505%	1697.8621%
2	39	13.4086%	1117.3066%
3	26	5.3556%	1122.3924%
4	18	3.8186%	1106.2849%
5	17	4.0384%	1483.3112%
6	6	2.0916%	598.1072%
7	4	0.7806%	663.0602%
8	4	0.7233%	761.152%
9	6	31.7091%	1240.0467%
10	2	0.0059%	441.502%

Table 5.1: Sequence length, occurences and their probabilities

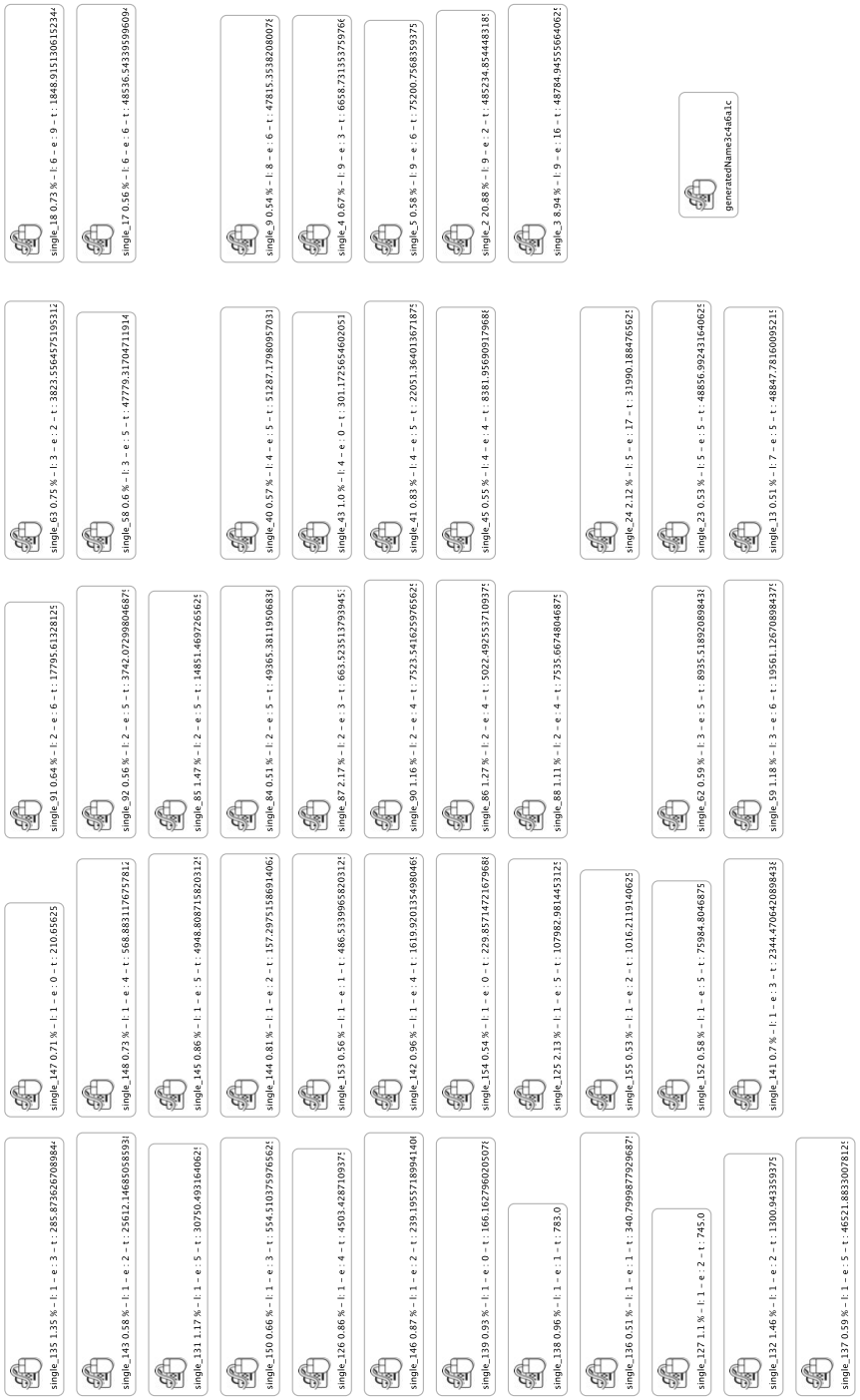


Figure 5.9: Single sequences > 0.5% overview

5.5 Optimization of Interaction Sequences

This section sketches optimization potentials of single interaction sequences that are extracted from all found interaction sequences of the process model. Figure 5.11 shows the single interaction sequence navigating test specifications in the Eclipse web browser. The complete interaction sequence consists of nine mouse wheel interactions. The probability of a mouse wheel interaction on the browser is 20.88% (of all interactions), whereby the browser is used with a probability of 23.69%. The sequence is not connected with the start interaction but with four other sequences. Interestingly the last interaction has a 80.94% chance to return to the first wheel interaction of the sequence. The complete sequence takes six seconds and has a high relevance. The sequence switches with 82.78% to another sequence and with a probability of 0.37% leaves the session.

The four input sequences are mainly triggered by the sequence itself, a 'click' sequence on the browser and two sequences with a high probability to be executed after a test execution took place. Figure 5.10 shows one of the two interaction sequences triggering the test spec review of figure 5.11. The colored element with a graph picture in the

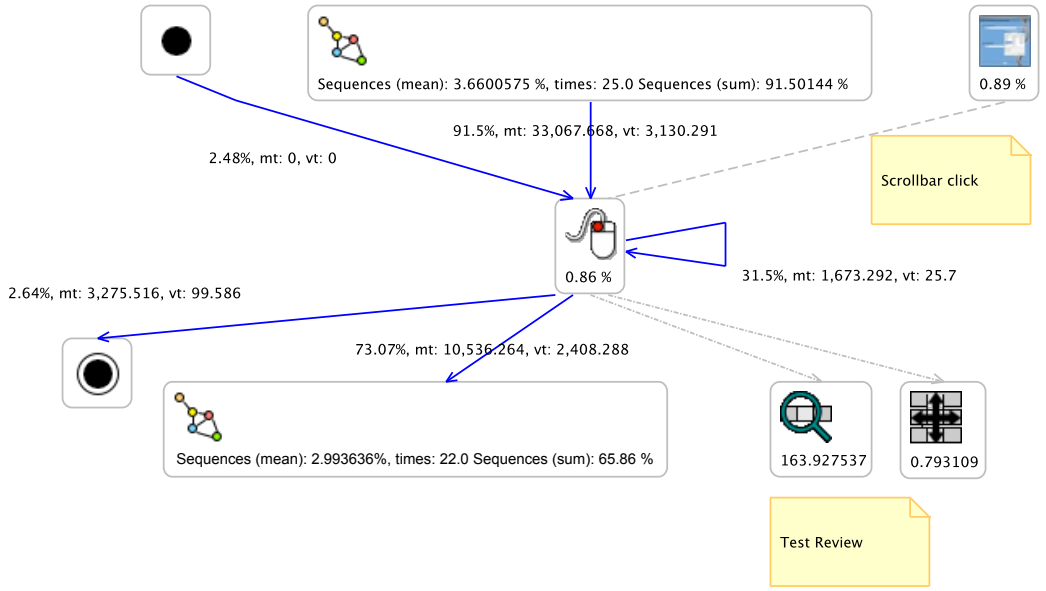


Figure 5.10: Test review as main external input sequence

top and in the bottom of figure 5.10 is an aggregation of 25 sequences (each time). The inputs have a probability of 3.66% each to start the interaction sequence and a

probability of 2.922% of leaving to other sequences in the mean. The number 163.92 describes one effect of the test review results. In this case its the average number of no value differences (successful - previously edited - test cells) during a MIL test.

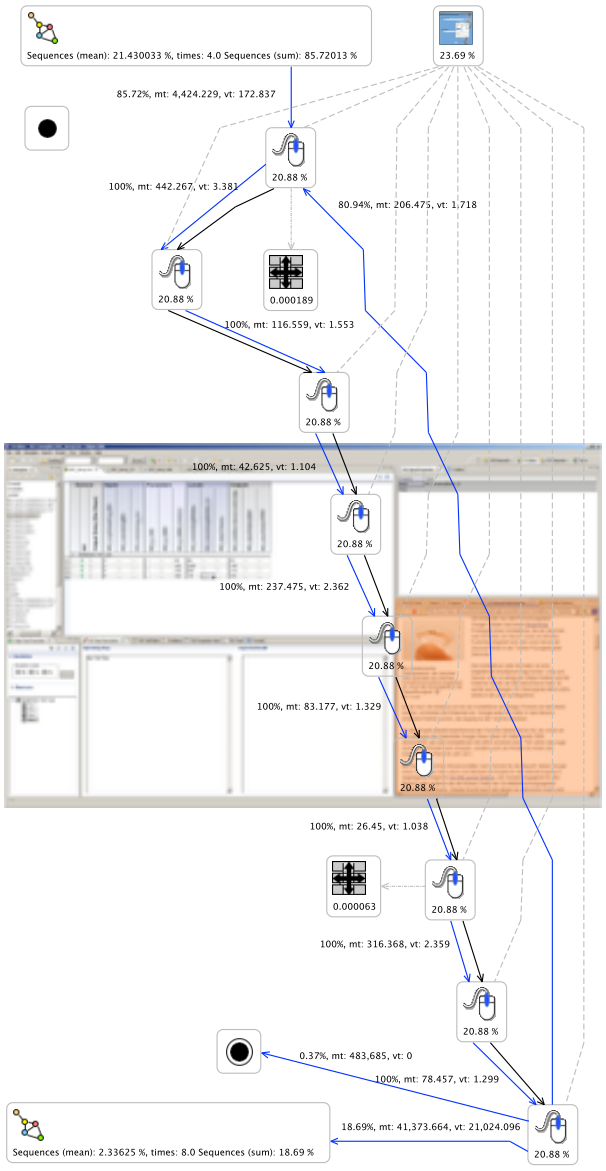


Figure 5.11: Navigation: test specification browser

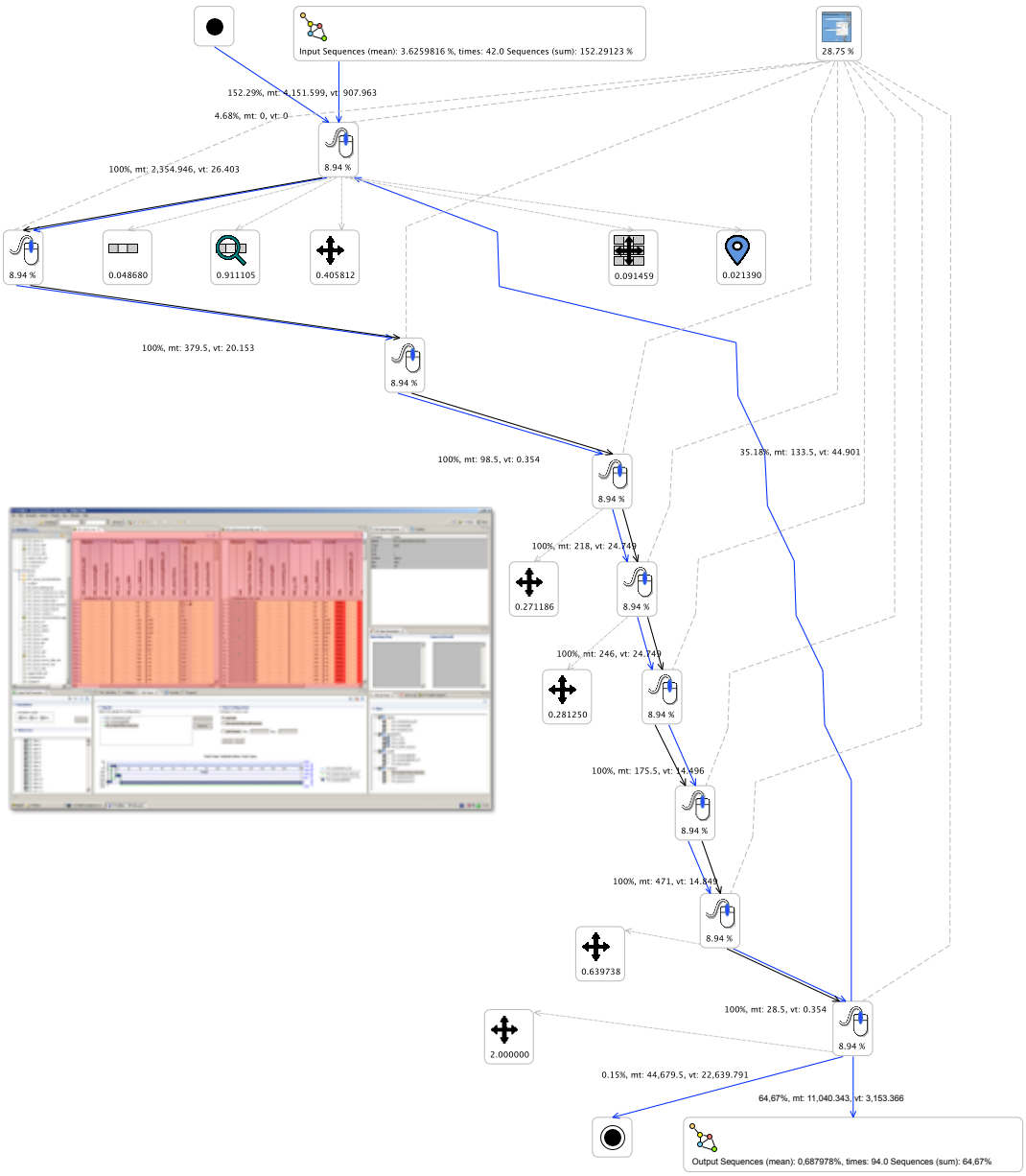


Figure 5.12: Navigation: test vectors

Figure 5.12 shows a similar navigation sequence on the test vectors. The average time spent to navigate in the test vectors is low compared to the test specifications and the significance of the sequences is less than 50% of the significance of the TVE

specifications. Both sequences have a high significance (spec browsing 80.94%, test vectors 35.18%) to be triggered recursively.

A software optimization regarding the navigation activities could be realized by a linkage between test vectors and test specifications. This linkage (possibly 1-n, hopefully 1-1) could be introduced in previous development steps or during the testing activity. The linkage should be changeable by the tester but is likely to be fix. Figure 5.13 shows the idea graphically. A reduction of the navigation interactions of

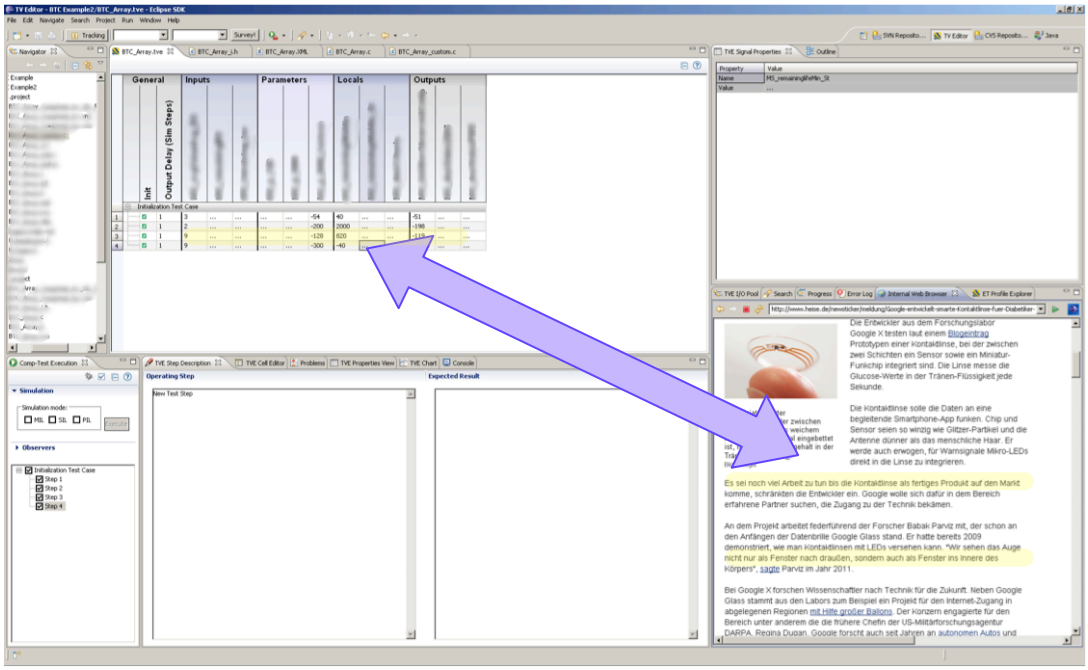


Figure 5.13: Optimization TVE navigation

sequence 5.11 (by re-linking input and output of the sequence) could reduce the time required for re-testing by approx. 1%. The effects increase at non re-testing interactions to approx. 2,2%. One obvious cause is that the specifications are new to the tester thus the navigation interactions are increased the first time. The sequence of figure 5.11 is often triggered by sequence of figure 5.12. It will likely be useful to reflect a scrolling in the test specifications in the test cases if the links between test specifications and test vectors exist. The question is what causes the scrollings in the test vectors. Figure 5.14 shows all input sequences with $> 0.5\%$ coverage. The major input sequence "125" with a coverage of 2.13% triggers the navigation sequence in the TVE grid with 7.64%. Figure 5.15 shows sequence "125" which is mainly conducted after a test execution took place. The scrolling operations could be avoided by an automatic

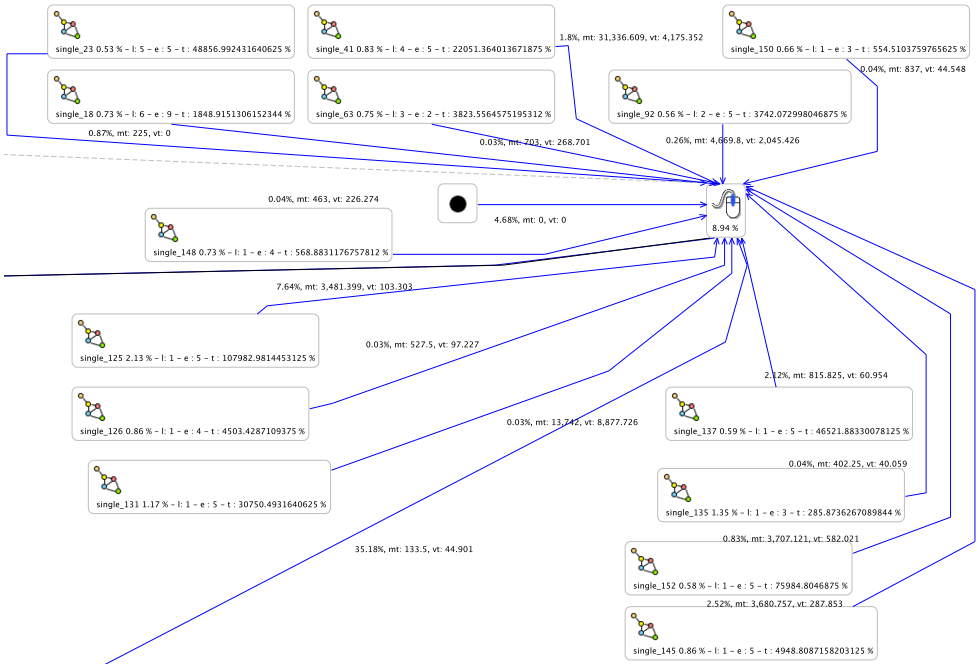


Figure 5.14: Input sequences of TVE grid navigation

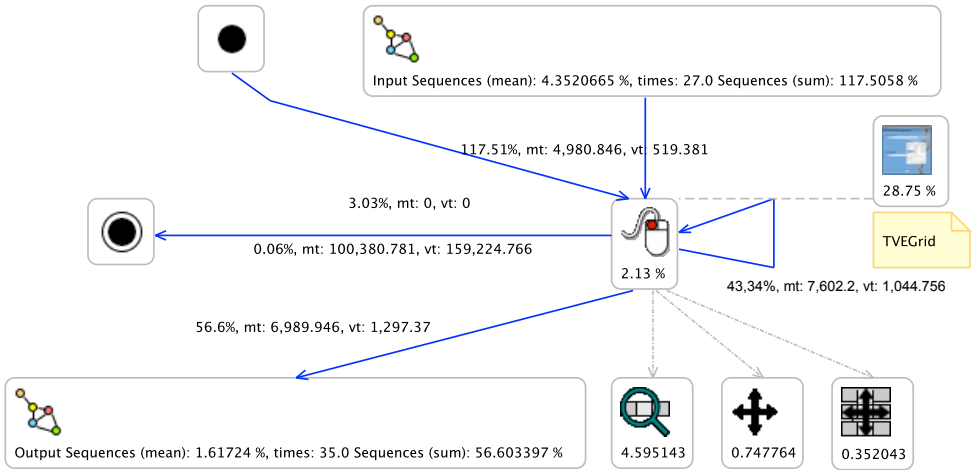


Figure 5.15: Input sequence '125' of TVE grid navigation

scrolling to the test results after a test execution was conducted. A linkage between test results and original vectors (like proposed for the test specifications) will likely reduce the required scrollings (at least when the results are represented in additional

result grids). A reduction of the TVE grid scrolling interactions of sequence 5.12 revealed approx. 10% time reduction for testing and re-testing interactions and there are other sequences showing additional potentials like `single_24`, `single_59` `single_85`, `single_86`, `single_90`. These are click sequences on the test vectors to scroll after a test execution took place - (presumably) to review the test results. The 10% is realized by removed the long scrolling sequence of 5.12. There exist shorter partial scrolling sequences with additional optimization potential. Another simple modification in this context would be "next/previous deviation in step/vector" macros to jump from one deviation to the next in the grid. Finally a linkage between test specifications and test vectors could also speed up the (correct) identification of the test vectors to be changed in the current testing activity.

The possible impacts were computed by simulation runs described in the following section 5.6. The reduction skips only navigation interaction of this sequence. It is assumed that after the specification part has been found the investigation takes place - not during the navigation. This can be assumed because the temporal deviations between the scrolling events of the sequence are below 0.5 seconds which is considered to be too short for an investigation of the test specification. Nevertheless only this (long) sequence was skipped. There are shorter scrolling sequences in the TVE sequence navigations and the optimization potential is large - in particular when errors are avoided that are introduced because the wrong specification section was chosen and other follow ups. Sequence `single_87` presented in figure 5.16 shows the (partial) dele-

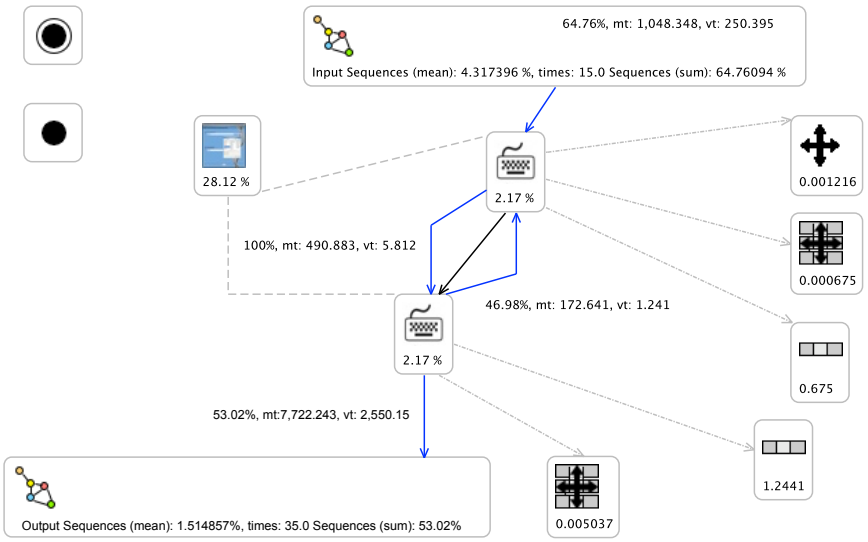


Figure 5.16: Delete value(s) sequence '87'

tion of test values. If the complete cell is deleted this could be optimized with a macro function. The remaining sequences that are found are of short length and equally low distributed regarding their coverage. The next optimization investigates probabilities between interaction sequences as shown in figure 5.17. Nearly all sequence transition

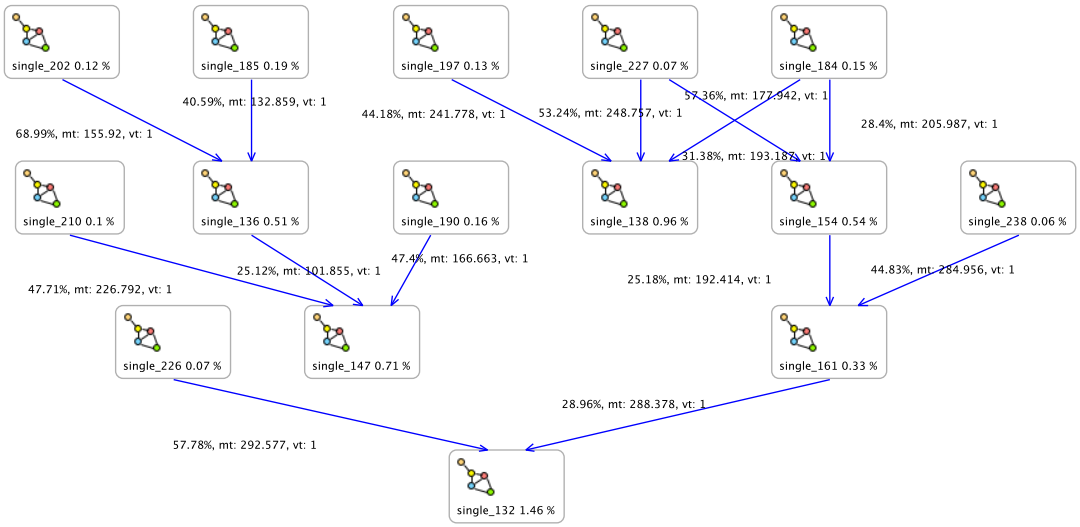


Figure 5.17: Probabilities between sequences

probabilities are low or the sequences itself have a low probability. Figure 5.18 shows three interaction sequence with a probability >= 1% and a transition probability >= 50%. The first sequence is a scrolling sequence on the sub-data view. The sub-data

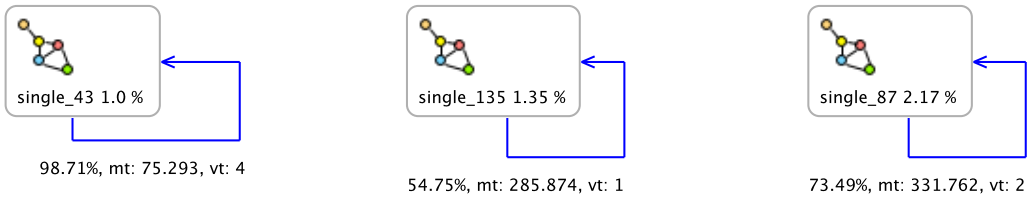


Figure 5.18: Three reflexive interaction sequences with a probability >= 1% and a transition probability >= 50%

view is an expansion of values of the TVE grid. One value of the TVE grid (presented as "...") is an array of values in the sub-data view. The handling of TVE grid and sub-data view is the same. The sequence is presented in figure 5.19. This is similar to the TVE Grid sequences and should be optimizable in the same way. The second

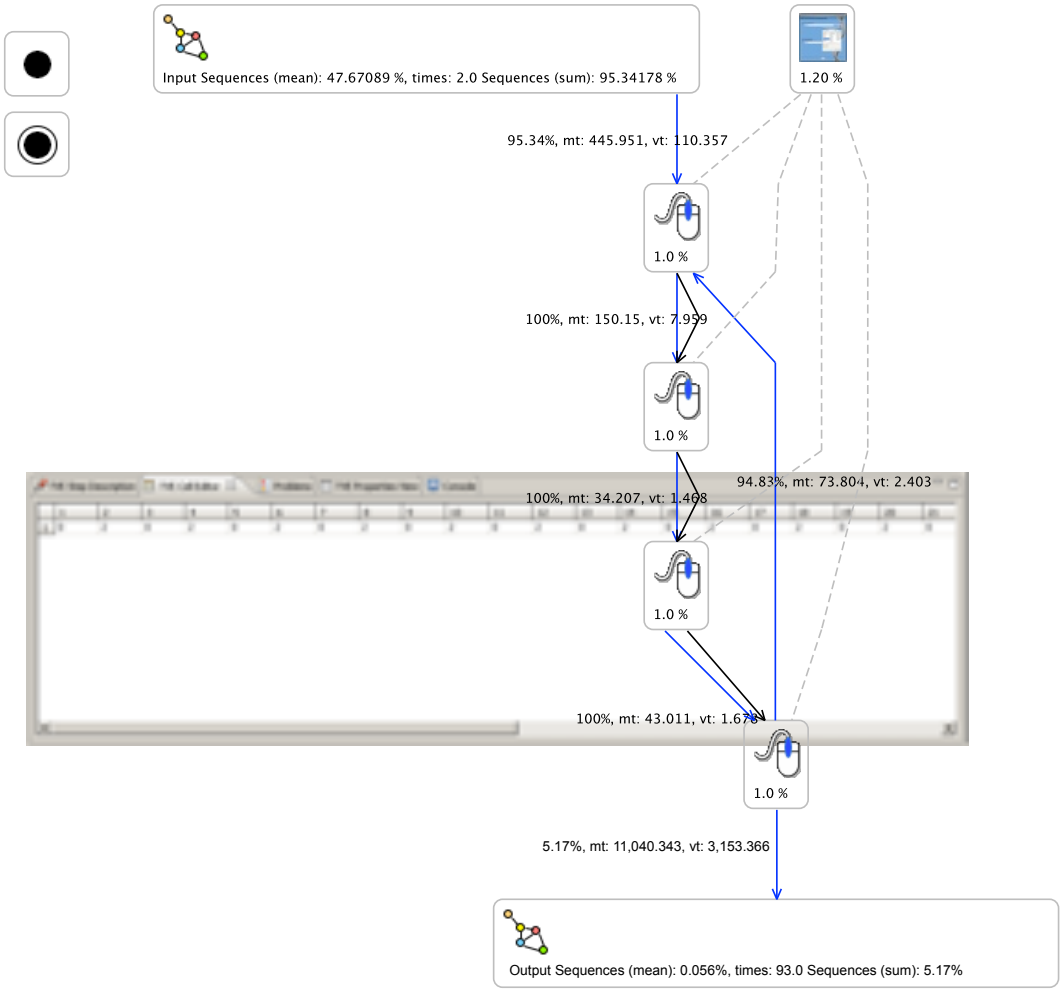


Figure 5.19: Sequence 43: scrolling sub-data view

sequence is key press “0” on the test vector grid inside a cell. A macro function to insert “0” could speed up this sequence. The last sequence is a backspace sequence shown in figure 5.20. This sequence can be optimized with a macro function like 5.16.

In addition to fix macro sequences the Test Vector Editor should support the creation and assignment of user-dependend keyboard macros to efficiently handle user-related and task-related editing operations effectively. In addition, some used-related macros could be computed by RMOF and to configure the Test Vector Editor before a test session.

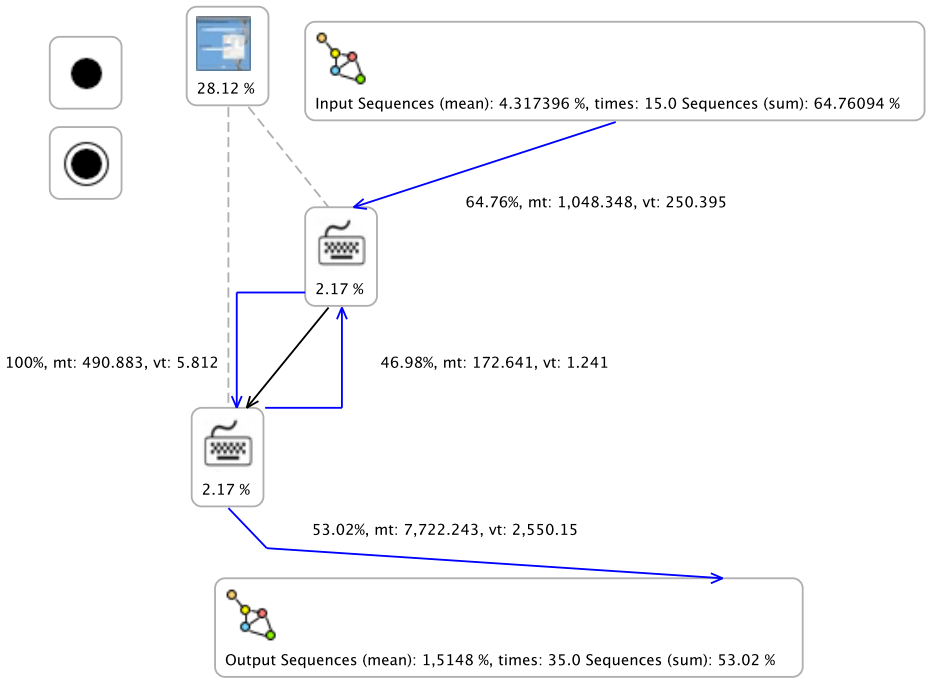


Figure 5.20: Sequence 87: backspaces in TVE Grid

The interaction sequence simulation is described in the next section.

5.6 Process Model Analysis: Simulation

The process simulation is realized as a concurrent Petri Net with probabilistic tokens based on section 3.18, page 132. Interactions are modeled as places, can have interaction effects and are related to GUI elements. Transitions have a probability in relation to the mean transition probabilities of all covered elements of the tracking database. Interaction effects are cumulated during a simulation run in the tokens with their probability. A simulation step concurrently switches a maximal subset of enabled tokens. The probability of each token is multiplied with the transition probability, if the token switches. If a token reaches a defined minimal probability, it is removed ('retained') from the interaction model. The token switching and the computation of the effects is done concurrently in the implementation. Figure 5.21 shows an example illustrating the simulation approach. Initially a green token is placed on the initial place (presented as a black filled circle). All outgoing transitions (with transition probabilities of 40%,

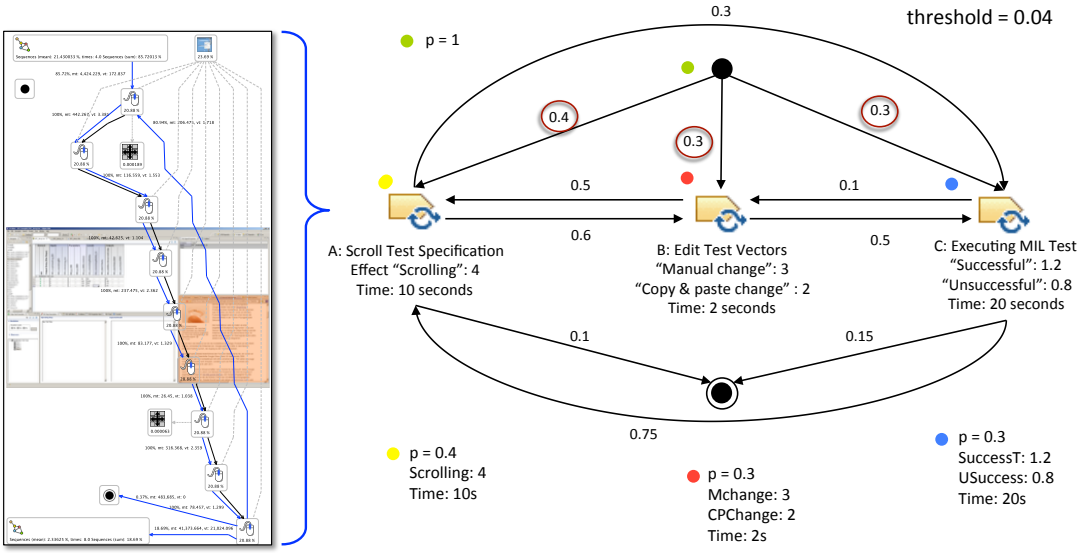


Figure 5.21: Simulation approach example

30%, and 30%) are enabled and fire once resulting in three tokens on the places A,B, and C respresenting different unconditional sequences. A presents a scrolling in the test specifications. B presents test vector editing operations. C represents the execution of a MIL test. The attributes of the tokens after the first switch are presented at the bottom of figure 5.21. including all seen interaction effects of the interaction sequences (e.g., a manual change of test vectors). Part of the blue token of step 2 in the simulation would not switch because the token probability if 30% and if multiplied with the transition probability of 10% (going to B) this would result in a token probability if 3% which is below the threshold probability of 4%. In this case a blue token "freezes" on C with a probability of 3%.

The ratio of the simulation approach is to execute all identified interaction sequences with their discovered probabilities as long as the probability of the sequence described by the token path is above a defined threshold. This results in a complete probabilistic approximation of all sequences seen in the tracking down to a certain minimal probability.

The expected values and the standard deviation of all retained tokens are computed in each simulation step until all token are retained. Expected value and standard deviation are the values in the first line. The second line sets the expected value in relation to the time and computes how much the value changes in 60s and how much time is required to change the value by one. The results were simulated on interaction pattern

with a coverage of 97.4831%. First, the unmodified interaction model was simulated, then the optimized interaction model was simulated to compute the effects of the optimizations.

The result discussion starts with detected scrollings. The expected time between scrollings is approx. four minutes. This is indirectly reflected in the view changes of the visible TVE grid. The visible TVE grid view changes approx. once every minute due to move and resize operations:

```
TVEGridVisible: 61.431058, std: 16.123318
[in 60s: 0.772253057725615, o: 01m 17s 694ms]
MoveX: 43.801703, std: 9.510480
[in 60s: 0.5506335159211536, o: 01m 48s 965ms]
MoveY: 27.758492, std: 7.324738
[in 60s: 0.34895345996474625, o: 02m 51s 942ms]
ResizeX: 33.408033, std: 9.133302
[in 60s: 0.4199741430423046, o: 02m 22s 865ms]
ResizeY: 16.350475, std: 5.024496
[in 60s: 0.20554268598278513, o: 04m 51s 910ms]
```

It will be interesting to investigate if a larger screen size (multiple screens) change this behavior. In particular regarding resize operations. The suggested scrolling optimization of section 5.5, page 172 will also be reflected here.

Every 18 minutes a menu is used. The concrete interaction sequences inform about the menu, that is was used, e.g., a rename, move, or import operation. The effects are additionally tracked for example in TVE grid changes or additional views opened.

The main (and rather abstract) information concerns the number of tests executions. Every 1 hours 40 minutes a test was executed in the sequences, whereby a re-test takes place in a similar time period. The tester can conduct three kinds of test variants (MIL, SIL, PIL) together in one test execution. The results are differentiated in result columns.

```
TestExecuted: 0.788331, std: 0.222333
[in 60s: 0.00991014794545417, o: 01h 40m 54s 400ms]
TestExecuted-TestREExecuted: 0.804766, std: 0.205024
[in 60s: 0.010116754652864606, o: 01h 38m 50s 755ms]
```

The Test Vector Editor is able to conduct MIL, SIL, PIL tests. The tracked information contains only MIL and SIL tests. The interactions done before a test execution

was computed, are summarized and classified into e.g., manual change operations (values are directly changed with keyboard interactions in the TVE grid), copy & paste operations or color changes. A color change can be done by the user to highlight cells or can be part of a test execution to visualize (dependent) tested test vectors/steps.

All changes are differentiated into RowsColumns or Rows. The first one represents single cells, the last one complete rows. Sometimes blocks are added representing a consecutive number of test rows.

```
changedBlocksBeforeMIL: 0.599558, std: 0.160439
[in 60s: 0.0075370705031842225, o: 02h 12m 40s 652ms]
changedRowsBeforeMIL: 4.570666, std: 1.258517
[in 60s: 0.05745808580920809, o: 17m 24s 239ms]
changedRowsColumnsBeforeMIL: 14.913078, std: 3.950378
[in 60s: 0.1874730940145881, o: 05m 20s 45ms]
```

In relation to the previous test execution values, it shows that a single MIL test normally tests approx. 15 new/changed values in approx. five rows in the average. Now the interactions are classified into manual editing interactions and aggregated copy/paste operations. A copy/paste operation is a copy and paste of a complete row in one operation to replicate one row into the next one.

```
copyAndPasteChangedRowsColumnsBeforeMIL: 0.227189, std: 0.060751
[in 60s: 0.002856002735193893, o: 05h 50m 08s 383ms]
copyAndPasteChangedRowsBeforeMIL: 0.093960, std: 0.025213
[in 60s: 0.0011811809832712142, o: 14h 06m 36s 618ms]
copyStepChangedRowsColumnsBeforeMIL: 5.589963, std: 1.480830
[in 60s: 0.07027172058595166, o: 14m 13s 828ms]
copyStepChangedRowsBeforeMIL: 0.806943, std: 0.214252
[in 60s: 0.010144122656024601, o: 01h 38m 34s 754ms]
deleteStepChangedRowsColumnsBeforeMIL: 1.723281, std: 0.460876
[in 60s: 0.02166345484892456, o: 46m 09s 641ms]
deleteStepChangedRowsBeforeMIL: 0.334686, std: 0.089814
[in 60s: 0.004207357230474926, o: 03h 57m 40s 733ms]
insertStepChangedRowsColumnsBeforeMIL: 6.111888, std: 1.619725
[in 60s: 0.07683286520900655, o: 13m 915ms]
insertStepChangedRowsBeforeMIL: 0.875479, std: 0.232058
[in 60s: 0.011005690427698528, o: 01h 30m 51s 725ms]
manualChangedRowsColumnsBeforeMIL: 0.285885, std: 0.075737
[in 60s: 0.003593873557278108, o: 04h 38m 15s 78ms]
manualChangedRowsBeforeMIL: 0.188524, std: 0.050532
```

```
[in 60s: 0.00236994550936315, o: 07h 01m 57s 37ms]
manualDeleteChangedRowsColumnsBeforeMIL: 0.186340, std: 0.052923
[in 60s: 0.002342486775172783, o: 07h 06m 53s 805ms]
manualDeleteChangedRowsBeforeMIL: 0.893187, std: 0.253930
[in 60s: 0.01122830367255548, o: 01h 29m 03s 638ms]
pasteStepChangedRowsColumnsBeforeMIL: 0.891480, std: 0.236453
[in 60s: 0.011206845693713692, o: 01h 29m 13s 870ms]
pasteStepChangedRowsBeforeMIL: 0.155885, std: 0.041294
[in 60s: 0.001959641781044828, o: 08h 30m 17s 840ms]
unknownBlocksChangedBeforeMIL: 0.026663, std: 0.007538
[in 60s: 3.351820878816066E-4, o: ---]
```

The most likely interaction is a copy/insert interaction followed by manual editing operations. The same information are summarized and analyzed for Software in the Loop test executions.

```
changedBlocksBeforeSIL: 0.511297, std: 0.136515
[in 60s: 0.006427535317477127, o: 02h 35m 34s 837ms]
changedRowsColumnsBeforeSIL: 15.330471, std: 4.060610
[in 60s: 0.1927201542737282, o: 05m 11s 332ms]
changedRowsBeforeSIL: 2.076008, std: 0.549927
[in 60s: 0.026097600015880757, o: 38m 19s 61ms]
```

It demonstrates that it's less likely to change complete blocks before a SIL test is done compared to a MIL test. The rest of the values is quite similar.

```
copyAndPasteChangedRowsColumnsBeforeSIL: 0.112735, std: 0.030486
[in 60s: 0.0014171956368078512, o: 11h 45m 37s 132ms]
copyAndPasteChangedRowsBeforeSIL: 0.053761, std: 0.014247
[in 60s: 6.758332462035221E-4, o: ---]
copyStepChangedRowsColumnsBeforeSIL: 2.929287, std: 0.777290
[in 60s: 0.03682421672827174, o: 27m 09s 362ms]
copyStepChangedRowsBeforeSIL: 2.008050, std: 0.563057
[in 60s: 0.02524330426118349, o: 39m 36s 867ms]
deleteStepChangedRowsColumnsBeforeSIL: 2.805823, std: 0.750421
[in 60s: 0.03527215263210606, o: 28m 21s 58ms]
deleteStepChangedRowsBeforeSIL: 0.863389, std: 0.234668
[in 60s: 0.010853704271705665, o: 01h 32m 08s 66ms]
insertStepChangedRowsColumnsBeforeSIL: 8.289486, std: 2.195662
[in 60s: 0.10420756000324001, o: 09m 35s 773ms]
insertStepChangedRowsBeforeSIL: 0.970736, std: 0.257596
```

```
[in 60s: 0.012203169800670254, o: 01h 21m 56s 755ms]
manualChangedRowsColumnsBeforeSIL: 0.254820, std: 0.067656
[in 60s: 0.003203351924356871, o: 05h 12m 10s 380ms]
manualChangedRowsBeforeSIL: 0.330646, std: 0.090991
[in 60s: 0.004156566490145358, o: 04h 34s 991ms]
manualDeleteChangedRowsColumnsBeforeSIL: 0.100721, std: 0.028509
[in 60s: 0.0012661723631193467, o: 13h 09m 46s 913ms]
manualDeleteChangedRowsBeforeSIL: 0.094624, std: 0.026775
[in 60s: 0.0011895176279797539, o: 14h 40s 614ms]
pasteStepChangedRowsColumnsBeforeSIL: 0.824555, std: 0.218529
[in 60s: 0.010365520150300934, o: 01h 36m 28s 421ms]
pasteStepChangedRowsBeforeSIL: 0.121194, std: 0.032515
[in 60s: 0.0015235369605064847, o: 10h 56m 22s 44ms]
unknownBlocksChangedBeforeSIL: 0.003774, std: 0.001007
[in 60s: 4.7444005093178736E-5, o: 13ms]
```

The largest difference shows a deviation to use copy/paste operations. In SIL testing the operation is only rarely used. After a test execution took place the results are parsed. The parsing takes into consideration what was changed on the test grid before a test was executed and differentiates direct changes and indirect (a change in a dependent test cell that was not changed before the test execution but changed by the test execution). NoValue differences indicate that the output vector of a previously changed input vector matches the test results.

```
noValueDifferencesMIL: 455.774487, std: 118.398683
[in 60s: 5.7295650707449255, o: 10s 471ms]
noValueDifferencesDependentMIL: 25.253322, std: 6.416270
[in 60s: 0.31746083924276813, o: 03m 08s 999ms]

noValueRowDifferencesMIL: 10.227078, std: 2.731456
[in 60s: 0.12856513777975806, o: 07m 46s 689ms]
noValueRowDifferencesDependentMIL: 0.707969, std: 0.190550
[in 60s: 0.008899910408887832, o: 01h 52m 21s 640ms]

valueDifferencesMIL: 34.633886, std: 9.075570
[in 60s: 0.43538440236808285, o: 02m 17s 809ms]
valueDifferencesDependentMIL: 2.347846, std: 0.606019
[in 60s: 0.029514889708611852, o: 33m 52s 872ms]

valueRowDifferencesMIL: 2.407073, std: 0.613960
[in 60s: 0.030259442644822394, o: 33m 02s 852ms]
```

```
valueRowDifferencesDependentMIL: 4.026180, std: 0.992508
[in 60s: 0.05061332491896774, o: 19m 45s 458ms]
```

Every 7m 46s a MIL vector is tested successfully. Every 33m 2s a MIL vector test fails.

```
noValueDifferencesSIL: 169.394603, std: 43.910020
[in 60s: 2.129468465625207, o: 28s 176ms]
noValueDifferencesDependentSIL: 5.645568, std: 1.456639
[in 60s: 0.07097073474133643, o: 14m 05s 418ms]

noValueRowDifferencesSIL: 3.802279, std: 0.987206
[in 60s: 0.04779864589605562, o: 20m 55s 265ms]
noValueRowDifferencesDependentSIL: 0.249593, std: 0.069415
[in 60s: 0.0031376423902990743, o: 05h 18m 42s 638ms]

valueDifferencesSIL: 41.374312, std: 10.707748
[in 60s: 0.520118657244326, o: 01m 55s 358ms]
valueDifferencesDependentSIL: 3.414305, std: 0.889456
[in 60s: 0.04292140769733019, o: 23m 17s 903ms]

valueRowDifferencesSIL: 3.961081, std: 1.024616
[in 60s: 0.04979496016522125, o: 20m 04s 941ms]
valueRowDifferencesDependentSIL: 0.760739, std: 0.191042
[in 60s: 0.009563295617400368, o: 01h 44m 33s 987ms]
```

Every 20m 55s a SIL vector is tested successfully. Every 20m 04s a SIL vector test fails. MIL tests happen more frequently. MIL testing is faster and therefore presumably chosen more often. It is more likely, that a (more precise) SIL test is executed if the MIL test failed. The maximal time duration of the longest sequence was 01h 19m which matches the average session duration of 1h 30m quite good. The question is if there are conditions that change the interaction sequences reflected in the simulation results. The next section will discuss this question by introducing a first idea of model complexity.

5.7 Process Model Impacts: Guards

The process models contain two 'types' of transitions. Unconditional ones with a probability of 100% and conditional ones with a probability <100%. By adding transition guards the uncertainty is further reduced. Guard sources are e.g., environment attributes (like display size), the behavior of the tester (like views opened and closed),

inputs (like Matlab Simulink models) or outputs (like test vectors). The usefulness of a guard source is related to its ability to prognosticate development activities/interactions. Therefore activity outputs are ignored.

Matlab Simulink models are developed before they are tested. A significant relationship of the models to the conducted test interactions would be useful to predict test interactions. The hypothesis is that the complexity of the Matlab Simulink model has an influence on the required test interactions. Silnab Models were introduced in section 5.1. They are an abstraction of Matlab Simulink models without concrete functions. Metrics and partial computations (simulating data flows) were used to formulate different ideas of testing complexity. The (assumed) reason is that the testing complexity is related to the different blocks building the system and the transitions between those components. The tester needs to control/understand the overall function composed by the blocks realizing the system including their temporal dimension. Therefore reflexive transitions are often of particular interest. The hypothesis is that this kind of complexity metric has an influence on the interaction sequences the tester uses.

Halstead metrics [122] were applied to count different Simulink blocks, inputs and outputs, or vocabularies (different Simulink blocks). These metrics are refined by adding some simulation semantics describing the relationships/data flows between Simulink blocks to compute, e.g., the cohesion of Simulink blocks. The models are (discretely)

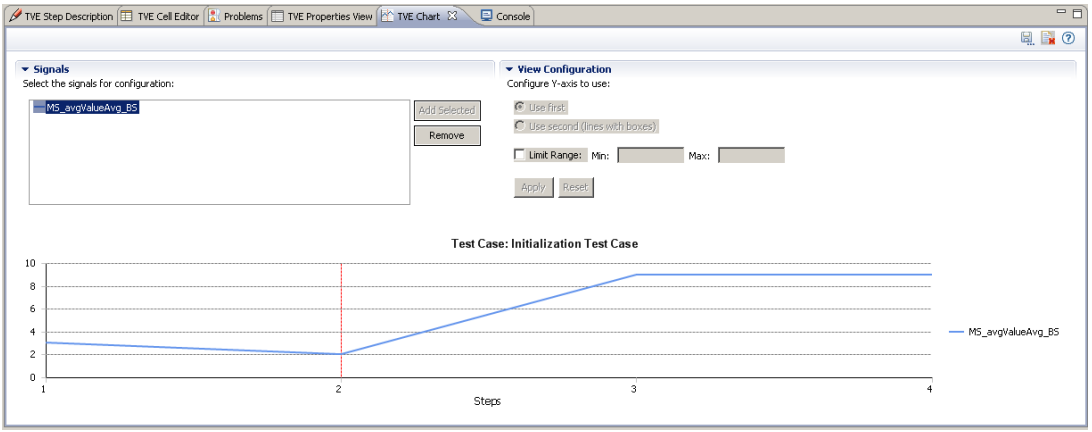


Figure 5.22: TVEChart / signal propagation view

executed. Silnab models abstract from the concrete function, therefore this kind of over-approximation gives an idea on how many functions and blocks are involved to compute the overall block function. Additionally, the screen size can be used by a

guard and the currently open views like a TVE signal view as presented in figure 5.22. In both cases the hypothesis is, that the screen size and the opened views have an effect on the interactions required. The display sizes used (in the guards) were “1280x1024”, “1562x922”, “1680x1050”.

The guards are generated by the GP framework and describe (an abstraction of) Silnab semantics. Basic block of each guard filter Silnab blocks by using attribute/value pairs. The filter can be easily extended and currently used the following attributes:

- type: the Simulink block type like Inport, Constant, Reference, Terminator, Ground, OR_STATE, ...
- subsystemlevel: if the system is composed of sub-systems, this variable shows the level of the sub-system of the Simulink block
- inputs, outputs: number of inputs and outputs. Input range is 0-32, output range is 0-92
- variable characteristics: type (e.g., double), scope (e.g., INPUT_DATA), and default value together with a scrambled ID
- expressions: number of comparisons, conjunctions, disjunctions is counted
- expressions-events: events produced and consumed is counted
- expressions-variables: variables assigned and referenced is counted
- values of the so-called property array: like OpenFunction, CopyFunction, DeleteFunction (in scrambled form)

The filter computes the number of blocks for each file matching the attribute/value pairs. The maximum number of Simulink blocks in one file was 3500. The blocks can be combined by applying the set operations join, cut, and complement. After the set operations were applied, the block filtering is complete. Additional guard components allow the computation of transitions between blocks to compute (transitively) reachable components including a cycle detection.

A final guard consists of a set of basic blocks and a relation specifying thresholds as upper or lower limit of the blocks counted. The blocks are previously computed for all Silnab files and the results are stored in a metrics database. The composed guards are evaluated with the enriched process models (see 5.2, page 161) .

The fitness function initially determines the set of conditional transitions and then tries to distinguish these transitions by applying the generated guards. Each guard

is applied to all interactions in the model and assigned $\frac{1}{(\text{mapped transitions})^2}$ fitness points. The guard with the highest fitness points wins an interaction. The winning guard of an interaction is removed from the set of competing guards. A removed guard is not longer considered in the fitness evaluation for the interaction won. The overall fitness value equals to the sum of all winning guards for all interactions.

The interactions model of section 5.6 was used to add guards. The model contained 9416 conditional transitions with a mean probability of 2.49%. The theoretical optimum of transitions that could be distinguished with assigned model files was approx. 50% which is equal to 4757 transitions. The GP algorithm required approx. three days with 300 nodes to find a fitness of 4606.9844 covering all transitions (by additionally using the display size and the open views). Table 5.2 shows the guard distribution of 70 guards.

Transitions covered size	Number of transitions	probability
1	4123	123.64%
2	897	37.67%
3	322	18.96%
4	151	10.55%
5	102	10.88%
6	41	6.48%
7	33	5.42%
8	25	4.16%
9	18	2.59%
10	6	0.775%
11	11	4.51%
12	5	1.70%
13	3	1.48%
14	1	0.52%
15	7	1.89%
16	3	0.82%
17	1	0.81%
19	1	0.06%
20	1	0.61%
23	1	0.38%
25	1	0.41%
29	1	0.67%

Table 5.2: Distribution of transitions with guards

Six guards with the highest impacts were extracted and simulated to determine the effects mainly regarding the time required to conduct a (re-)test and the time to test a vector or test step successful or unsuccessful. Four guards showed positive effects (the required times were reduced) when models with the matching criteria are avoided. The comparison with section 5.6 shows significant effects. First of all, all scrollings are avoided by 25%, whereby scrolling effects emerge more drastically in the TVE Grid. These operations less often conducted.

```
moveX: 12.865223, std: 3.389716
[in 60s: 0.17570412980451214, o: 05m 41s 483ms]
-moveY: 16.675269, std: 3.836564
[in 60s: 0.22773904484846708, o: 04m 23s 459ms]
resizeX: 12.113256, std: 3.543305
[in 60s: 0.1654342980932856, o: 06m 02s 681ms]
resizeY: 11.335867, std: 3.048035
[in 60s: 0.15481726308213106, o: 06m 27s 553ms]
```

The scrollings are reduced up to 25%. This has a significant effect on the time required to conduct re-tests.

```
TestExecuted-TestREExecuted: 1.543419, std: 0.431370
[in 60s: 0.021078928570989364, o: 47m 26s 444ms]
```

The time required to re-test is reduced to approx. 47%. Initial tests require approx. the same period of time. The changes done before testing are done faster and in larger units.

```
changedBlocksBeforeMIL: 0.621282, std: 0.170153
[in 60s: 0.008485034116115925, o: 01h 57m 51s 273ms]
changedBlocksBeforeSIL: 0.548754, std: 0.150289
[in 60s: 0.0074944904858203175, o: 02h 13m 25s 881ms]
changedRowsColumnsBeforeMIL: 17.943288, std: 4.914191
[in 60s: 0.24505675623463855, o: 04m 04s 841ms]
changedRowsColumnsBeforeSIL: 18.466041, std: 5.057359
[in 60s: 0.25219614617372743, o: 03m 57s 910ms]
changedRowsBeforeMIL: 6.53321, std: 1.32334
[in 60s: 0.05745808580920809, o: 13m 24s 239ms]
changedRowsBeforeSIL: 2.497445, std: 0.683984
[in 60s: 0.034108341986590275, o: 29m 19s 100ms]
```

The MIL test results show all a temporal reduction. This includes the required time resulting in a row difference, albeit the row differences on dependent MIL rows is significantly increased.

```
noValueDifferencesMIL: 710.034635, std: 199.159452
[in 60s: 9.69715135755854, o: 06s 187ms]
noValueDifferencesDependentMIL: 30.489574, std: 8.368531
[in 60s: 0.4164050568944878, o: 02m 24s 90ms]

noValueRowDifferencesMIL: 16.131863, std: 4.565161
[in 60s: 0.2203175805588982, o: 04m 32s 334ms]
noValueRowDifferencesDependentMIL: 0.729045, std: 0.199523
[in 60s: 0.009956786436760083, o: 01h 40m 26s 40ms]

valueDifferencesMIL: 53.825487, std: 14.813534
[in 60s: 0.7351104729642756, o: 01m 21s 620ms]
valueDifferencesDependentMIL: 3.858975, std: 1.051179
[in 60s: 0.05270315234713727, o: 18m 58s 451ms]

valueRowDifferencesMIL: 3.945968, std: 1.080907
[in 60s: 0.053891240792271825, o: 18m 33s 353ms]
valueRowDifferencesDependentMIL: 0.249024, std: 0.066724
[in 60s: 0.003400988568134905, o: 04h 54m 01s 929ms]
```

Every 4m 32s a MIL vector is tested successfully (58.3% of the time previously required). Every 18m 2s a MIL vector test fails (54,6% of the time previously required).

```
noValueDifferencesSIL: 248.177132, std: 69.031191
[in 60s: 3.3894279232596696, o: 17s 702ms]
noValueDifferencesDependentSIL: 8.079400, std: 2.194429
[in 60s: 0.11034274258561193, o: 09m 03s 760ms]

noValueRowDifferencesSIL: 5.338710, std: 1.435640
[in 60s: 0.07291232850197123, o: 13m 42s 906ms]
noValueRowDifferencesDependentSIL: 0.123306, std: 0.033239
[in 60s: 0.0016840236660742626, o: 09h 53m 48s 952ms]

valueDifferencesSIL: 72.996585, std: 20.181075
[in 60s: 0.996935775434972, o: 01m 184ms]
valueDifferencesDependentSIL: 4.537232, std: 1.244025
[in 60s: 0.061966303313006794, o: 16m 08s 268ms]
```

```
valueRowDifferencesSIL: 5.002094, std: 1.372367
[in 60s: 0.06831506834632936, o: 14m 38s 283ms]
valueRowDifferencesDependentSIL: 0.319025, std: 0.086354
[in 60s: 0.004357012383423392, o: 03h 49m 30s 904ms]
```

Every 13m 42s a SIL vector is tested successfully (65,49%). Every 14m 38s a SIL vector test fails (72,92%). The four guards contain only scrambled functions, which is interesting because there exist various attributes like subsystem level, variables types, number of comparisons that are not used by the GP algorithm to generate the guards.

```
#[PreSaveFcn = c9a1ae815ea1853d48a49be852f488b42b34cf8351c7]
#[InitFcn = 90ba0d13369e9aae16f9d2c5972983734916926e075d509]
#[PostLoadFcn = 416c6ea0b101b39afe389622a980c1cad5fe7964cf8]
```

These components and their connections (input, output, and cyclomatic complexity) are computed transitively and build the guards. Each of the four guards filters approx. 20 of the 60 model files. The testing complexity comes mainly from certain Simulink block functions and the dataflows between components. Three guards compute a cyclic complexity of some previously filtered components (filtered by their function) and then compute all input or output transitions (over x components) to the (transitive) reflexive components.

```
<=(#it[ #cc[ #[LoadFcn = ....], 9], 777)
```

The remaining guard components filter different Simulink blocks. The guards currently used to distinguish interaction sequences have a high impact on the conducted interactions and are composed of some Matlab metrics, display properties and open views. The last two had no significant impact on the simulation results. Open views used in the guard with a negative effect were the “PackageExplorer”, “TaskList”, or “ConsoleView”. But these are common views - not special views of the Test Vector Editor - therefore ignored.

The tracking sequences were filtered regarding the model files the guards covered. Then the tracking database and the interaction sequences were recomputed. Figure 5.23 presents the single sequences overview.

The coverage computed of the interaction sequences model was >99%. The two long scrolling sequences are still found (compare figure 5.9, page 171) but their coverage is changed. The browser specification navigation coverage of all scrollings in the test specifications increases slightly from 20.88% to 21.07%. The TVE specification coverage of all scrollings in the test vectors decreases significantly from 8.95% to 7.01%. The

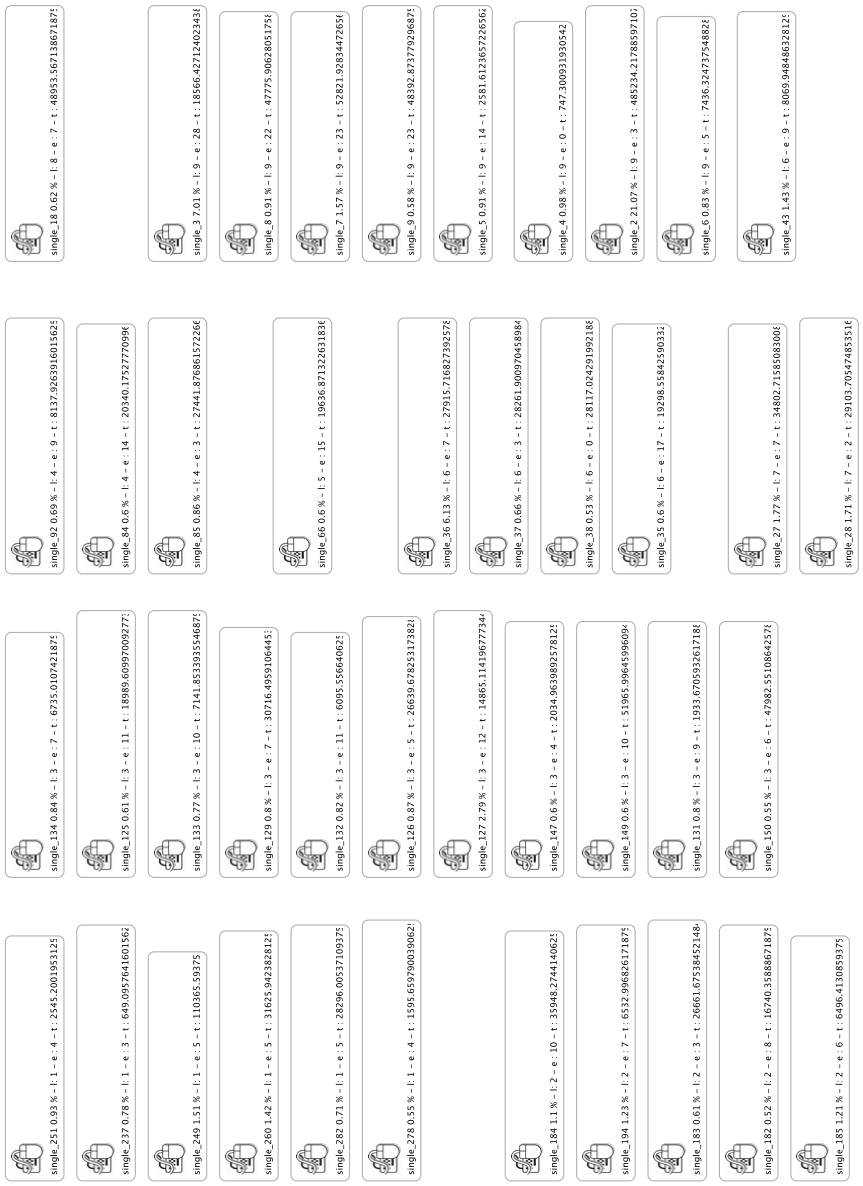


Figure 5.23: Single interaction sequences (without guard(ed) Silnab files)

last sequence has a high impact on the interaction effects and an obvious explanation is that the reduced model "complexity" results in less scrolling operations. The results reveal that there were optimization potentials even for a well understood, small testing context and a highly optimized tool that supports the testing activities very well.

The synthesis approach uses heuristics. Hence the question of the method validity should be addressed (asking the question if the same results are found when the method is reapplied). In this case the method validity is related to the coverage of all found interaction sequences. The error probability or the chance of sequence deviations is therefore 100% - the computed coverage. In other words by increasing the coverage the deviation probability is reduced. Therefore the coverage was increased to 99% in the last computation run to decrease the chance that the optimized sequence still exists but is not found by the heuristics. In the worst case the sequence is found with 1% less coverage. Although this is inlikely due to deterministic components in the evaluation (e.g., that prefer always longer sequences). Another simple solution would be to set all random numbers seeds to the same numbers of the previous computation run or simply to search for the optimized pattern determistically in the tracked data.

6 Conclusion

RMOF was developed as a flexible process meta-modeling language to describe, synthesize and analyze process models with all their artifacts and activities including their dynamical dimensions. Such a language is required because existing languages and optimization methods are often not suitable and sustainable. In particular the data required to achieve this task needs to be tracked manually, often resulting in high effort and errors. This thesis addresses both aspects to establish formal but flexible process models including methods and techniques to synthesize and analyse them appropriately. These process models are precise enough to answer all kinds of development questions and to optimize activities and tools to guide the creation of artifacts optimally in relation to the defined process goals.

The evaluation demonstrated that it is possible to synthesize a process model as an abstraction of real testing tool interactions within an industrial context. The interactions were automatically tracked by additional software components that were introduced into the development environments of the tester. The tester only needed to start the tracking, which itself is completely automatic. Although the tracking components were enrichments of the testing software they are developed as general Eclipse plugins and are only very loosely coupled with the test software plugins. The extension of the tracking towards other Eclipse/Java based environments is an easy task and opens a wide range of data sources to be tracked automatically.

The evaluation revealed optimization potentials in the interaction sequences of the tester that can be utilized by changes in the test tool software and reduce the required test effort effectively. It revealed impacts on the computed process model in terms of the input model semantics. The input model semantics were based on the usage of certain functions in Simulink blocks and the data flow between some of the components. If these functions are used and the blocks are connected transitively with a certain amount of other blocks, the interactions of the tester change including their outcome, and the test effort increases. This information can be used to guide testing activities and optimize the testing tool further.

The synthesized process model, its optimization and possible impacts are shown exemplarily. Other hypotheses can be checked with RMOF and different models and viewpoints can be computed (e.g., a click model to optimize mouse interactions or

model semantics that induce test errors with a high(er) probability). The current optimizations focus the software (GUI layout, macros, (additional) views) and the semantics of the models but can be easily extended towards, e.g., developer dependend attributes like tool knowledge or environment attributes like noise level.

The approach was evaluated with an industrial partner and tracked data based on real testing activities of several months with a group of different testers. It required various optimizations to synthesize matching interaction models with a significant coverage but then the tracking data of one person year was analyzed with a few hundred computing nodes in some hours. Nevertheless the degree of freedom of a tester conducting the testing activities is limited compared to other development tasks (e.g., debugging of the Matlab Input models when a model error was detected). It can be assumed that the methods and techniques developed in this thesis have the potential to investigate all kinds of development activities and artifacts. This includes methods (e.g., test-driven development), techniques (e.g., refactorings), development languages (e.g., Java) and additional information sources (e.g., user eye tracking to analyze focus patterns). This will offer the possibility to understand, control and optimize methods and techniques on a new level. The RMOF implementation allows the formal synthesis, visualization and analysis of significant process elements in relation to defined process goals.

Currently, the process synthesis is done with Genetic Programming. Concerning the inherently unbounded model space that needs to be explored this will likely not change. Nevertheless there are promising optimizations regarding for example the preanalysis/computation of the tracking database. Some general optimizations like caching are already implemented, others will likely improve the convergence of the GP computations further e.g., model fitting with RMOF. The model fitting will presumably benefit from a co-evolutionary GP approach that exploits the flexibility of RMOF to formulate model classification/fitting criteria. The process analysis is currently done with a simulation based approach. There exist various formal methods and techniques (e.g., probabilistic model checking) that will likely increase the precision and might have the possibility to offer additional insights into the data.

This work is a first step towards a new kind of process modeling and optimization targeting an industrial context. It holds the promise to introduce methods and techniques into the development of computer systems that have the same level of maturity as methods and techniques of traditional engineering disciplines while still providing the suitability and sustainability that is required to apply and optimize them in a controlled way.

Bibliography

[1] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, and T. Wolf, "Model Checking (Grundlagen und Praxiserfahrungen).," Informatik Spektrum, vol. 27, no. 2, pp. 146–158, 2004.

[2] R. Koppe, A. Koppe, and R. Buschermöhle, "Ergebnisse der Studie CONTROL," tech. rep., OFFIS - Institut für Informatik, Oldenburg, Germany, 2011.

[3] R. Buschermöhle, H. Eekhoff, H. Frommhold, B. Josko, and M. Schiller, SUCCESS - Erfolgs- und Misserfolgsfaktoren bei der Durchführung von Hard- und Softwareentwicklungsprojekten in Deutschland. Oldenburg, Germany: BIS-Verlag der Carl von Ossietzky Universität, 2011.

[4] R. Buschermöhle, H. Eekhoff, H. Frommhold, B. Josko, and M. Schiller, "SUCCESS - Erfolgs- und Misserfolgsfaktoren bei der Durchführung von Hard- und Softwareentwicklungsprojekten in Deutschland - erweiterte Analyse -," tech. rep., OFFIS - Institut für Informatik, Oldenburg, Germany, 2011.

[5] V-Modell XT v1.4, 2012. Available online at `http://www.v-modell-xt.de` last visited on 11-11-2014.

[6] Object Management Group, SPEM - Software Process Engineering Metamodel v2.0, 2008. Available online at `http://www.omg.org/spec/SPEM/` last visited on 11-11-2014.

[7] K. Jensen, Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Berlin, Germany: Springer-Verlag, 1997.

[8] G. Junkermann, B. Peuschel, and W. Schaefer, "Merlin: Supporting cooperation in software development through a knowledge-based environment," in Proceedings of Software Process Modelling and Technology (J. Kramer and B. Nuseibeh, eds.), (New York City, New York, United States), pp. 103–129, John Wiley & Sons, Inc., 1994.

[9] W. F. Tichy, "A data model for programming support environments and its application," in Trends in Information Systems (B. Langefors, A. A. Verrijn-Stuart, and G. Bracchi, eds.), pp. 219–236, Amsterdam, The Netherlands: North-Holland Publishing Co., 1986.

[10] S. Bandinelli, C. Ghezzi, A. Fuggetta, and L. Lavazza, "Spade: An environment for software process analysis, design, and enactment," in Proceedings of Software Process Modeling and Technology, (New York City, New York, United States), pp. 223–248, John Wiley & Sons, Inc., 1994.

[11] B. Kiepuszewski, Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003.

[12] H. Zhang, B. Kitchenham, and D. Pfahl, "Reflections on 10 years of software process simulation modelling: A systematic review," in Proceedings of the International Conference on Software Process, (Leipzig, Germany), Springer-Verlag, 2008.

[13] G. Rozenberg, ed., Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, (River Edge, NJ, USA), World Scientific Publishing Co., Inc., 1997.

[14] S. J. Mellor and M. Balcer, Executable UML: A Foundation for Model-Driven Architectures. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[15] R. Buschermöhle and J. Oelerink, "Rich meta object facility as formal integration platform: Syntax, Semantics, and Implementation," Innovations in Systems and Software Engineering, vol. 4, no. 3, pp. 249–257, 2008.

[16] L. Osterweil, "Software processes are software too," in Proceedings of the 9th international conference on Software Engineering, ICSE '87, (Los Alamitos, CA, USA), pp. 2–13, IEEE Computer Society Press, 1987.

[17] C. A. Petri, Kommunikation mit Automaten. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, 1962. Schriften des IIM Nr. 2, 68 pages.

[18] W. M. Zuberek, "Performance evaluation of concurrent systems using timed petri nets," in Proceedings of the 1985 ACM thirteenth annual conference on Computer Science, (New York, New York, USA), pp. 326–329, ACM, 1985.

[19] R. Valk, "Self-modifying nets, a natural extension of petri nets.," Lecture Notes in Computer Science: Automata, Languages and Programming, vol. 62, pp. 464–476, 1978.

[20] "Integrated DEFinition for function modeling (IFDEF0)," 1993. Available online at `http://www.idef.com/pdf/idef0.pdf` last visited on 11-11-2014.

[21] K. E. Huff and V. R. Lesser, "A plan-based intelligent assistant that supports the software development," in Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, SDE 3, (New York, New York, USA), pp. 97–106, ACM, 1988.

[22] G. E. Kaiser, P. H. Feiler, and S. S. Popovich, "Intelligent assistance for software development and maintenance," IEEE Software, vol. 5, pp. 40–49, May 1988.

[23] T. Katayama, "A hierarchical and functional software process description and its enaction," in Proceedings of the 11th international conference on Software engineering, ICSE '89, (New York, New York, USA), pp. 343–352, ACM, 1989.

[24] W. Deiters and V. Gruhn, "Managing software processes in the environment MELMAC.," SIGSOFT Software Engineering Notes, vol. 15, pp. 193–205, Dec. 1990.

[25] A. Bröckers, M. C. Lott, H. D. Rombach, and M. Verlage, "MVP-L: Language report version 2," tech. rep., AG Software Engineering, Kaiserslautern, Germany, 1995.

[26] H. D. Rombach, "MVP-L: A language for process modeling in-the-large," tech. rep., University of Maryland at College Park, College Park, MD, USA, 1991.

[27] M. Baldi, S. Gai, M. L. Jaccheri, and P. Lago, "Object Oriented Software Process Model Design in E3," Software Process Modelling and Technology, Research Studies Press, pp. 279–290, 1994.

[28] M. Jaccheri and R. Conradi, "Techniques for process model evolution in epos," in Proceedings of IEEE Transactions on Software Engineering, vol. 19, pp. 1145 – 1156, 1993.

[29] R. Greenwood, I. Robertson, J. Sa, B. Warboys, R. A. Snowdon, and R. F. Bruynooghe, "PADM: Towards a Total Process Modelling System," Software Process Modelling and Technology, 1994.

[30] J. Sa and B. Warboys, Specifying Concurrent Object-based Systems Using Combined Specification Notations. Manchester UMCS, University of Manchester, Department of Computer Science, 1991.

[31] G. Canals, N. Boudjlida, D. Jean-Claude, C. Godart, and J. Lonchamp, ALF: a framework for building process-centred software engineering environments, pp. 153–185. Taunton, UK: Research Studies Press Ltd., 1994.

[32] W. Emmerich and V. Gruhn, "Funsoft nets: A petri-net based software process modeling language," in Proceedings of the sixth international workshop on software specification and design, pp. 175–184, IEEE Computer Society Press, 1996.

[33] S. M. Sutton, Jr., and L. J. Osterweil, "The design of a next-generation process language," in Proceedings of 6TH European Software Engineering Conference and the 5th ACM Sigsoft Symp. on the foundations of Software Engineering, (Berlin, Germany), pp. 142–158, Springer-Verlag, 1997.

[34] S. Taft, R. Duff, R. Brukardt, E. Ploedereder, P. Leroy, and E. Schonberg, Ada 2012 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/2012 (E). Lecture Notes in Computer Science / Programming and Software Engineering, Berlin, Germany: Springer-Verlag, 2014.

[35] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and J. Stanley M. Sutton, "Using Little-JIL to Coordinate Agents in Software Engineering," in Proceedings of Automated Software Engineering Conference (ASE 2000), Grenoble, France, pp. 155–163, IEEE, 2000.

[36] J.-Y. J. Chen, "CSPL: An Ada95-Like, Unix-Based Process Environment," IEEE Transactions Software Engineering, vol. 23, pp. 171–184, March 1997.

[37] S. Dami, J. Estublier, and M. Amiour, "APEL: a Graphical Yet Executable Formalism for Process Modeling," Automated Software Engineering, vol. 5, pp. 61–96, 1997.

[38] R. Solingen and E. Berghout, The Goal/Question/Metric method: a practical guide for quality improvement of software development. New York, New York, USA: McGraw-Hill, 1999.

[39] J. M. Ribó and X. Franch, "Building Expressive and Flexible Process Models Using a UML-Based Approach," in Proceedings of Software Process Technology: 8th European Workshop, EWSPT 2001, (Witten, Germany), pp. 152–172, Springer-Verlag, 2000.

[40] D. Rossi and E. Turrini, "EPML : Executable Process Modeling Language - UBLCS-2007-22," tech. rep., Department of Computer Science, University of Bologna, 2007.

[41] Object Management Group, MOF - Meta Object Facility Core Specification Version 2.0, 2011. Available online at `http://www.omg.org/cgi-bin/doc?formal/2006-01-01` last visited on 11-11-2014.

[42] Object Management Group, Object Constraint Language v2.4, 2014. Available online at `http://www.omg.org/spec/OCL/` last visited on 11-11-2014.

[43] A. Agrawal, "Metamodel based model transformation language to facilitate domain specific model driven architecture," in Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, (Anaheim, CA, USA), pp. 118–119, ACM, 2003.

[44] E. C. Kabore, Contribution à l'automatisation d'un processus de construction d'abstractions de communication par transformations successives de modèles. PhD thesis, Université Rennes 1, 2008.

[45] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "VIATRA: Visual automated transformations for formal verification and validation of UML models," in Proceedings of the 17th IEEE International Conference on Automated Software Engineering (J. Richardson, W. Emmerich, and D. Wile, eds.), (Edinburgh, UK), pp. 267–270, IEEE Press, September 23–27 2002.

[46] K. Backhaus, B. Erichson, W. Plinke, and W. Weiber, Multivariate Analysemethoden. Eine anwendungsbezogene Einführung. Berlin, Germany: Springer-Verlag, 2006.

[47] P. P. Eckstein, Angewandte Statistik mit SPSS. Praktische Einführung für Wirtschaftswissenschaftler. Wiesbaden, Germany: Gabler Verlag, 2004.

[48] D. Luenberger and Ye, Linear and Nonlinear Programming. Berlin, Germany: Springer-Verlag, third ed., 2008.

[49] A. Pnueli, "The temporal logic of programs," in Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1977.

[50] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Transactions on Programming Languages and Systems, vol. 8, pp. 244–263, Apr. 1986.

[51] E. M. C. Jr., O. Grumberg, and D. A. Peled, Model Checking. The MIT Press, 1999.

[52] W. van der Aalst, A. Weijter, and L. Maruster, "Workflow mining: Discovering process models from event logs," IEEE Transactions on Knowledge and Data Engineering, vol. 16, pp. 1128–1142, 2003.

[53] Object Management Group, CWM - Common Warehouse Metamodel v1.1, 2003. Available online at `http://www.omg.org/spec/CWM/` last visited on 11-11-2014.

[54] Object Management Group, Unified Modeling Language 2.0 Infrastructure Specification, 2007. Available online at `http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/` last visited on 11-11-2014.

[55] Object Management Group, Unified Modeling Language 2.0 Superstructure Specification, 2007. Available online at `http://www.omg.org/spec/UML/2.0/Superstructure/PDF/` last visited on 11-11-2014.

[56] R. Eshuis and R. Wieringa, "A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models," Tech. Rep. TR-CTIT-01-04, University of Twente, Centre for Telematics and Information Technology, Enschede, The Netherlands, 2001.

[57] H. Störrle, "Semantics and Verification of Data Flow in UML 2.0 Activities," Electronic Notes in Theoretical Computer Science, vol. 127, pp. 35–52, Apr. 2005.

[58] H. Störrle, "Semantics of UML 2.0 Activities," in Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, (Piscataway, NJ, USA), IEEE Press, 2004.

[59] W. Damm, B. Josko, A. Pnueli, and A. Votintseva, "A discrete-time UML semantics for concurrency and communication in safety-critical applications," in Proceedings of Science of Computer Programming, vol. 55, (Amsterdam, The Netherlands), pp. 81–115, Elsevier Science Publishers B. V., 2005.

[60] D. Akehurst and S. Kent, "A relational approach to defining transformations in a metamodel," in UML 2002 - The Unified Modeling Language (J.-M. Jézéquel, H. Hussmann, and S. Cook, eds.), vol. 2460 of Lecture Notes in Computer Science, pp. 243–258, Berlin, Germany: Springer-Verlag, 2002.

[61] G. Engels, R. Heckel, and J. M. Küster, "Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model," Lecture Notes in Computer Science, vol. 2185, p. 272ff., 2001.

[62] D. Varró and A. Pataricza, "VPM: Mathematics of metamodeling is metamodeling mathematics," Journal of Software and Systems Modeling, vol. 1, pp. 1–24, 2003.

[63] M. Alanen and I. Porres, "Subset and Union Properties in Modeling Languages," tech. rep., Abo Akademi University Department of Information Technologies, Turku, Finland, 2005.

[64] "The Eclipse Modeling framework." Available online at `http://www.eclipse.org/modeling/emf` last visited on 11-11-2014.

[65] D. Steinberg, F. Budinski, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework. Eclipse Series, Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., v2 ed., 2008.

[66] XMI - XML Metadata Interchange Specification v1.2, 2002. Available online at `http://www.omg.org/cgi-bin/doc?formal/2002-01-01` last visited on 11-11-2014.

[67] "Java Emitting Templates." Available online at `http://www.eclipse.org/modeling/m2t/downloads/?project=jet` last visited on 08-13-2009.

[68] "Acceleo - a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard." Available online at `https://eclipse.org/acceleo/` last visited on 11-11-2014.

[69] "MOF Model to Text Transformation language (MOFM2T) v1.0," 2008. Available online at `http://www.omg.org/spec/MOFM2T/1.0/` last visited on 11-11-2014.

[70] "Xpand - a statically-typed template language." Available online at `http://wiki.eclipse.org/Xpand` last visited on 11-11-2014.

[71] "EMF Model Query Project." Available online at `http://www.eclipse.org/modeling/emf/downloads/?project=query` last visited on 11-11-2014.

[72] "EMF Model Compare Project - compare and model EMF models." Available online at `https://www.eclipse.org/emf/compare/` last visited on 11-11-2014.

[73] "EMF Model Validation Project - checking constraints." Available online at `http://eclipse.org/modeling/emf/downloads/?project=validation` last visited on 11-11-2014.

[74] "Graphical Model Framework / Project." Available online at `http://www.eclipse.org/modeling/gmp/` last visited on 11-11-2014.

[75] R. C. Gronback, Eclipse Modeling Project : A Domain-Specific Language (DSL) Toolkit. Eclipse Series, Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2009.

[76] "EMF UML2 project." Available online at `http://www.eclipse.org/modeling/mdt/?project=uml2` last visited on 11-11-2014.

[77] "BPMN2 Business Process Model and Notation 2.0 project." Available online at `http://www.eclipse.org/modeling/mdt/?project=bpmn2` last visited on 11-11-2014.

[78] "IMM Information Metamodel project - provide metamodel/profile implementations based on the information management metamodel omg specification." Available online at `http://www.eclipse.org/modeling/mdt/?project=imm` last visited on 11-11-2014.

[79] "ATL - model transformation technology." Available online at `https://eclipse.org/atl/` last visited on 11-11-2014.

[80] "Declarative QVT." Available online at `http://www.eclipse.org/mmt/?project=qvto` last visited on 11-11-2014.

[81] "Operational QVT." Available online at `http://www.eclipse.org/modeling/m2m/downloads/index.php?project=qvtoml` last visited on 11-11-2014.

[82] "The Meta Object Facility (MOF) Query/View/Transformation language v2.0." Available online at `http://www.omg.org/cgi-bin/doc?ptc/2007-07-07` last visited on 11-11-2014.

[83] "Teneo project - a database persistency solution for EMF using Hibernate or EclipseLink." Available online at `http://www.eclipse.org/modeling/emft/?project=teneo` last visited on 11-11-2014.

[84] "Net4J - extensible client-server system based on the Eclipse Runtime and the Spring framework." Available online at `http://eclipse.org/modeling/emft/?project=net4j` last visited on 11-11-2014.

[85] "CDO - Connected Data Objects." Available online at `https://eclipse.org/cdo/` last visited on 11-11-2014.

[86] "Eclipse Process Framework." Available online at `http://www.eclipse.org/epf/` last visited on 11-11-2014.

[87] K. Schwaber, Agile Project Management with Scrum. Redmond, WA, USA: Microsoft Press, 2004.

[88] K. Beck and C. Andres, Extreme Programming Explained: Embrace Change (2nd Edition). Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[89] W. M. P. van der Aalst, J. Desel, and A. Oberweis, eds., Proceedings of Business Process Management, Models, Techniques, and Empirical Studies, vol. 1806 of Lecture Notes in Computer Science, (Berlin, Germany), Springer-Verlag, 2000.

[90] A. Yakovlev, L. Gomes, and L. Lavagno, Hardware Design and Petri Nets. Berlin, Germany: Springer-Verlag, 2000.

[91] P. Lutz and W. Harro, Theoretische Informatik: Petri Netze. Berlin, Germany: Springer-Verlag, 2002.

[92] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[93] S. Luke, Essentials of Metaheuristics. Department of Computer Science, George Mason University: Lulu, 2.0 ed., 2013. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[94] K. De Jong, "Evolutionary computation: A unified approach," in Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '08, (New York, New York, USA), pp. 2245–2258, ACM, 2008.

[95] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies: A comprehensive introduction," Natural Computing, vol. 1, pp. 3–52, May 2002.

[96] L. J. Fogel, Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming. New York City, New York, United States: John Wiley & Sons, Inc., 1999.

[97] J. H. Holland, Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Cambridge, Massachusetts, USA: MIT Press, 1975.

[98] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). Cambridge, Massachusetts, USA: A Bradford Book, 1 ed., Dec. 1992.

[99] L. Altenberg, "The evolution of evolvability in genetic programming," in Advances in Genetic Programming (K. E. Kinnear, ed.), pp. 47–74, Cambridge, Massachusetts, USA: MIT Press, 1994.

[100] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin, Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[101] D. Sankoff and J. B. Kruskal, Time warps, string edits, and macromolecules. Cambridge, England: Cambridge University Press, 2000.

[102] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, pp. 671–680, 1983.

[103] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in Proceedings of the 1st International Conference on Genetic Algorithms, (Hillsdale, NJ, USA), pp. 183–187, L. Erlbaum Associates Inc., 1985.

[104] F. Gruau, "Genetic synthesis of boolean neural networks with a cell rewriting developmental process," in Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92) (J. D. Schaffer and D. Whitley, eds.), pp. 55–74, IEEE Computer Society Press, 1992.

[105] D. J. Montana, "Strongly typed genetic programming," Evolutionary Computation, vol. 3, pp. 199–230, 1994.

[106] H.-P. Schwefel and G. Rudolph, "Contemporary evolution strategies," in Proceedings of the Third European Conference on Advances in Artificial Life, (London, UK), pp. 893–907, Springer-Verlag, 1995.

[107] D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in Proceedings of the Third International Conference on Genetic Algorithms, pp. 116–121, Morgan Kaufmann, 1989.

[108] J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, Massachusetts, USA: MIT Press, 1994.

[109] A. Brindle and U. of Alberta. Department of Computing Science, Genetic Algorithms for Function Optimization. Technical report (University of Alberta. Department of Computing Science), University of Alberta, 1981.

[110] D. Whitley, J. Kauth, and C. S. U. D. of Computer Science, GENITOR: A Different Genetic Algorithm. Technical report (Colorado State University. Department of Computer Science), Colorado State University, Department of Computer Science, 1988.

[111] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach," IEEE Transactions on Evolutionary Computations, vol. 3, pp. 257–271, Nov. 1999.

[112] S. Baluja, "Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning," tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[113] H. Mühlenbein, "The equation for response to selection and its use for prediction," Evol. Comput., vol. 5, pp. 303–346, Sept. 1997.

[114] G. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," in Proceedings of IEEE Transactions on Evolutionary Computation, pp. 523–528, IEEE Computer Society Press, 1998.

[115] Z. Skolicki, "An analysis of island models in evolutionary computation," in Proceedings of the 2005 workshop on Genetic and evolutionary computation, GECCO '05, (New York, New York, USA), pp. 386–389, ACM, 2005.

[116] Z. Skolicki, "An Analysis of Island Models in Evolutionary Computation". PhD Dissertation, George Mason University, 2007, 2007.

[117] D. J. Montana, "Strongly typed genetic programming," Evolutionary Computation, vol. 3, pp. 199–230, June 1995.

[118] "SWT: Standard Widget Toolkit." Available online at `http://www.eclipse.org/swt/` last visited on 11-11-2014.

[119] "Usage Data Collector." Available online at `http://www.eclipse.org/epp/usagedata/` last visited on 09-14-2009.

[120] J. J.H. Morris and V. Pratt., "A linear pattern-matching algorithm," tech. rep., University of California, Berkeley, California, USA, 1970.

[121] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. New York, New York, USA: Cambridge University Press, 2002.

[122] M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series). New York, New York, USA: Elsevier Science Inc., 1977.

List of Tables

List of Figures