



Co Wolf

A Co-Evolution Tool

Entwicklungsprojekt Nr. 3

Co-Evolution of Architectural and Certification Models

Christian Karl Bernasko, Manuel Borja, Verena Käfer,
David Krauss, Michael Müller, Philipp Niethammer, Tim
Sanwald, Jonas Scheurich, David Steinhart, Rene Trefft,
Johannes Wolf, Michael Zimmermann

Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Lars Grunske
Supervisor:	Sinem Getir
Commenced:	2014-04-28
Completed:	2014-10-28
CR-Classification:	D.2.2, D.2.11

Abstract

Models are a great aid to reduce the complexity of the software system so that analysis tools and humans can conceive it. To regard the different aspects of a software system, as for example architecture, performance or reliability, several models of the same system are needed, though. Transformations translate the information contained in one model to another model so that the user only has to complete information specific to that model type.

CoWolf delivers a framework for model development, transformation and analysis implementing the idea of incremental model transformation. Incremental transformation isolates changes that were done to a model to selectively transform only these which solves two problems of classic transformations in practice: The runtime of the algorithm to translate huge models even if the changes are minor and the need for the users to provide additional information over and over again.

Furthermore, we check our implementation and the idea of incremental transformation by evaluating the model versions of an evolving Pick & Place Unit according runtime and the need of user input for classical and incremental transformation.

Contents

1	Introduction	1
1.1	Goals	2
1.2	Document Structure	2
2	Related Work	5
2.1	Model Driven Development	5
2.2	Model Difference	6
2.3	Model Transformation	7
3	Project Management	9
3.1	Scrum	9
3.2	Team	11
3.3	Roles	11
3.4	Continuous Delivery	12
3.5	Tooling	18
4	Requirements	27
4.1	Customer	27
4.2	Infrastructure	28
4.3	Models	29
4.4	Analysis	30
4.5	Editors	31
4.6	Evolution	32
4.7	Co-Evolution	33
4.8	Maintenance Preparations	34
5	Foundations and Technologies	35
5.1	Eclipse-Plugins	35
5.2	Model Analyzer	42
5.3	Logback and SLF4J	45
6	Models	47
6.1	Software Architecture Models	47
6.2	Quality of Service Models	55

Contents

7	Transformations	67
7.1	State Chart to DTMC and Vice Versa	67
7.2	CTMC to DTMC and Vice Versa	68
7.3	Fault Tree to CTMC	69
7.4	Component Diagram to Fault Tree	71
7.5	Sequence Diagram to LQN	74
8	Architecture	79
8.1	Concept and Overview	79
8.2	Models	80
8.3	Evolution	82
8.4	Co-Evolution	84
8.5	Graphical Interface	85
9	Implementation	87
9.1	Models	87
9.2	Version Management	93
9.3	Analysis	95
9.4	Evolution	99
9.5	Co-Evolution Framework	103
9.6	Integration Testing	109
10	Acceptance Criteria	111
10.1	Format	111
10.2	Basic Actions	113
10.3	Model Editor	114
10.4	Test-Case Description and Execution	122
11	Evaluation	125
11.1	Goals	125
11.2	Design	126
11.3	Threats to Validity	127
11.4	Results	127
11.5	Conclusion	130
12	Developer Guide	133
12.1	Develop a New Model	133
12.2	Develop a New Evolution	137
12.3	Develop a new Co-Evolution	144

13 Use of Co Wolf	151
13.1 Installation	151
13.2 Create New Models	156
13.3 Export Models	159
13.4 Working with Versions	160
13.5 Evolution of a Model	162
13.6 Co-Evolve a Model	163
13.7 Analyze a Model	168
14 Future Work	171
14.1 Development	171
14.2 Functionality	171
15 Conclusion	175
Bibliography	177
Glossary	183
Acronyms	185

List of Figures

2.1	MDA models and transformations [Ley13]	6
3.1	CoWolf merge token and traffic light	14
3.2	Continuous delivery infrastructure	15
3.3	Registration of Extension Points	20
3.4	Example for an GitHub issue.	24
5.1	Eclipse Modeling Framework (EMF) editor with a <i>CTMC</i> meta model.	36
5.2	Side-by-side editing (textual and graphical)	38
5.3	Edit operation defined as Henshin rule	41
5.4	Processing pipeline of SiLift [Keh+14b].	41
5.5	XFTA model and script files	43
6.1	Overview of a component diagram. [Sd]	48
6.2	A component [Sd]	48
6.3	A provided Interface [Sd].	49
6.4	A required Interface [Sd].	49
6.5	A simple connector [Sd].	49
6.6	A multi connector [Sd].	49
6.7	Overview of a statechart.	50
6.8	A statemachine.	50
6.9	A state.	50
6.10	A composite state.	51
6.11	An entry state.	51
6.12	An exit state.	51
6.13	A do action.	51
6.14	A transition	51
6.15	Overview of a sequence diagram. [Sd]	52
6.16	A lifeline header [Sd]	52
6.17	A synchronous call with its reply message [Sd]	53
6.18	An asynchronous call [Sd]	53
6.19	A message which creates a new lifeline [Sd]	53
6.20	A message which deletes the receiver [Sd]	54
6.21	An execution on a lifeline with its start and end point [Sd]	54
6.22	A sub-sequence which is executed several times [Sd]	54
6.23	Two alternative behaviours depending on the condition [Sd]	55

List of Figures

6.24	The DTMC meta model	56
6.25	Graphical representation of a simple DTMC model in CoWolf.	57
6.26	The CTMC meta model.	58
6.27	Graphical representation of a simple CTMC model in CoWolf	58
6.28	Layered Queueing Network (LQN) example	59
6.29	Graphical representation of a processor	60
6.30	Graphical representation of a task	61
6.31	Graphical representation of an entry	61
6.32	Graphical representation of an activities graph	62
6.33	Example of a Fault Tree	62
6.34	Graphical representation of a hazard	63
6.35	Graphical representation of a basic event	63
6.36	Graphical representation of an undeveloped event	64
6.37	Graphical representation of an intermediate event	64
6.38	Graphical representation of an AND gate	64
6.39	Graphical representation of an OR gate	64
6.40	Graphical representation of an INHIBIT gate	65
6.41	Graphical representation of an XOR gate	65
6.42	Graphical representation of an PRIORITY-AND gate	65
7.2	And Gate	69
7.3	Or Gate	70
7.4	Priority And Gate	71
7.5	New component instances pattern	72
7.6	New connections pattern	73
7.7	New connection between sensor and software component pattern	74
7.8	All created tasks are related to the CPU processor.	75
7.9	Each lifeline will be transformed to a task.	75
7.10	Transformation of synchronous messages	76
7.11	Transformation of asynchronous messages	76
8.1	Overview of the components.	79
8.2	Important elements for the realization of the evolution feature of CoWolf	83
9.1	User interface of the EMF tree view editor	88
9.2	Basic structure of the .odesign file	88
9.3	Structure of the .odesign file for the Statechart model	90
9.4	User view of the edit palette for the fault tree model editor	90
9.5	Definition of the element creation in Sirius	92
9.6	ModelAssociationManager Model	94
9.7	Continuous Time Markov Chain (CTMC) properties wizard	96
9.8	Snippet of the .lqn file's template	97

List of Figures

9.9	UML class diagram of the <code>AbstractEvolutionManager</code> class.	100
9.10	UML class diagram of the <code>TechnicalDifferenceBuilder</code> class	101
9.11	The wanted structure of a transformation graph.	104
9.12	The graph structure after adding all resources.	105
9.13	Transformation Mapping Editor.	108
11.1	Time needed for execution in each transformation step.	128
11.2	Number of rules executed in each transformation step.	129
11.3	Manual adjustments needed after each transformation step	131
12.1	Ecore classes are inherit from <i>IDBase</i> in the <i>CommonBase</i> meta model	134
12.2	Import of a.ecore metamodel into an genmodel.	135
12.3	Definition of a new Model Extension	137
12.4	MANIFEST.MF → Dependencies for <code>EvolutionManager</code>	138
12.5	MANIFEST.MF → Extension for <code>EvolutionManager</code>	140
12.6	MANIFEST.MF → Dependencies for <code>TechnicalDifferenceBuilder</code>	140
12.7	Components of the transformation graph	146
12.8	Example of a mapping based transformation rule	147
12.9	Example of a transformation rule using SiLift Differences	148
13.1	Create a new CoWolf project	156
13.2	Create a new model	157
13.3	The structure of an aird file	157
13.4	An error occurs while opening an editor	158
13.5	An error occurs while renaming or moving model files	158
13.6	This dialogue shows up after editing model files	159
13.7	Wizard to export a CTMCs to a PRISM model.	160
13.8	Before applying a patch	161
13.9	After applying a patch	161
13.10	Model Difference Wizard of CoWolf	163
13.11	Calculated difference of two model versions.	164
13.12	Evolution view of an example model.	165
13.13	Co-Evolution Wizard of CoWolf	166
13.14	Co-Evolution Results View of CoWolf	167
13.15	Inconsistent co-evolution detection	167
13.16	Preferences view for CTMC	168
13.17	CTMC properties wizard	170

List of Tables

3.1 Team experience	11
10.1 Testcase matrix	123
11.1 Goal 1: Performance of co-evolution	125
11.2 Goal 2: Usability of co-evolution	126
11.3 Measurements for the performance of the transformation process (M1/M2/M3)	128
11.4 Most costly Henshin rule with total execution time and number of executions	129
11.5 Measurements for the manual amount of work needed after transformation .	130

Listings

3.1	Example of a plug-in manifest.	19
9.1	Example rule mapping.	107
9.2	SWTBOT example	109
12.1	Code to remove in the model wizard	136
12.2	Example EvolutionManager implementation for CTMC.	139
12.3	Example TechnicalDifferenceBuilder implementation.	141
12.4	Example SERGe configuration file.	143
13.1	Self signed cert import.	153

Introduction

Author: Philipp Niethammer

Models gain a steadily increasing importance in today's software systems. Often, great parts of programs are generated from domain specific models and analysis on performance, reliability and safety are completely done on separate models. Obviously, a model supporting all these duties would be very complex to the point where it is not usable anymore. It is desirable to split the information into different models that are all specialized for a specific task, for example by a well-founded theory on analysis methods and a rich tool infrastructure. For instance, it does not make sense to implement complex probabilistic analysis in a domain specific model, if the same task can be easily performed using Markov-Chains. Therefore, transformations between different types of models play a prominent role.

However, as program code does, models change over time, especially in iterative or agile software development where modifications are often minor but rather frequent. This *evolution* of a model often entails that other models must be updated as well. For example, if a new software component is added to the architecture, the performance model has to regard this component, too. Classic transformations can be not suitable for this use case, as they have two main drawbacks.

First, the steps of a transformation are typically proportional to the size of the models. Thus, the transformation of a big model can take a long time, even if there were only very small changes. Simple one-to-one transformations, which translate an element of the source model into one element of the target model, have a linear runtime, but there are complex scenarios of pattern creation that have exponential runtime. For these, a model instance quickly reaches a size where the transformation is not economically reasonable anymore.

Second, often the resulting model is not complete since the source model does not provide all needed information. These information must be supplied by the user after each transformation, that is investing manpower, hence increased costs.

A solution to these problems is seen in incremental transformation [JE04], a process that identifies changes done to the source model and translates only these changes to the target model instead of a full transformation. We describe the ongoing incremental transformation following the evolution as *co-evolution*.

1. Introduction

1.1 Goals

Author: Philipp Niethammer

CoWolf is an extensible tool for model evolution and co-evolution management. It comprises mainly three aspects:

▷ The management of associations between Model Instances

Different model instances often have associations in between them. Models of the same type can be predecessor or successors, performance models base on specific versions of architecture models. CoWolf stores and maintains those associations so that they are sound at any time and dependencies can be found and viewed easily.

▷ Deliver utilities for model development and analysis

For consistent development of the models, CoWolf provides a common environment with editors and verification for the different model types. Furthermore, interfaces to external tools are provided to analyze models.

When models get bigger, finding changes between versions of the same model is not trivial. Common difference detection tools (`diff`) mostly are not very usable for comparing models as they either work on a textual base (i.e. find the differences in the XML-File) or can only identify basic changes like additions or deletions. While the former case is at least quite usable, both results may be hard to interpret for more complex operations. By including SiLift [Keh+14a], CoWolf can find and view changes on a higher level. For example, if a method in a UML class model is moved to the classes' common parent (*pull up*), a simple `diff` returns multiple deletions and a creation of methods. SiLift identifies this change as a common refactoring measurement and reports it as such.

▷ The co-evolution of an associated model on the basis of evolutions

As described before, models may have to be updated if other models changed. Often, those updates can be described canonically. CoWolf features the definition of rules that define the relation between model types. Using these rules, co-evolutions can be done automatically for all associated models.

1.2 Document Structure

Author: Verena Käfer

First of all the related work will be discussed in Chapter 2. Then an overview about the project management and the used tools follows in Chapter 3. Afterwards Chapter 4 describes the necessary requirements for the project including the customer. Now the used foundations and technologies are explained in Chapter 5. The used models are described in Chapter 6 and the transformations between them in Chapter 7. Chapter 8 then shows the architecture of CoWolf and Chapter 9 shows the implementation of the project. Chapter

1.2. Document Structure

10 continues with the acceptance criteria. Afterwards the evaluation is shown in Chapter 11 and a developer and user guide is included in Chapter 12 and 13. Finally Chapter 14 describes the future work and Chapter 15 the conclusion.

Related Work

There is a lot of related work for model driven development, model differences and model transformations. The important parts for our project are described in the following chapter.

2.1 Model Driven Development

Author: Manuel Borja

Motivation

New technologies, programming languages and platforms evolve and get born constantly. That represents to the companies a difficulty to take a decision about which technology or set of technologies would be more suitable in order to implement and install its software system. Not only because the offer is high, but also because they have to support interoperability with another systems.

Instead of spending large amounts of time in technical issues, companies rather should be focused in analyzing and modeling their business logic independently in which platform or programming language the software system will be implemented. That is the core idea of MDA (Model-Driven Architecture) and MDD (Model-Driven Development) [MM01].

The main principle of MDA is the construction of a set of models which express the business logic and the latter transformation to the specific platform. The architecture of the MDA models is described in Figure 2.1

- ▷ Computation Independent Model (CIM): the system objects (persons, artifacts, interactions, etc) are represented as they are with no correspondences with the software system. It is also called domain model or business model.
- ▷ Platform Independent Models (PIM): represents the structure and functions of the system without technical details.
- ▷ Platform Specific Models (PSM): provides a view of the system from the platform specific point of view.

Transformations

Transformations allow the representation of the system in different levels of abstraction. They can also produce new artifacts, viewpoints or models based on patterns. In the case

2. Related Work

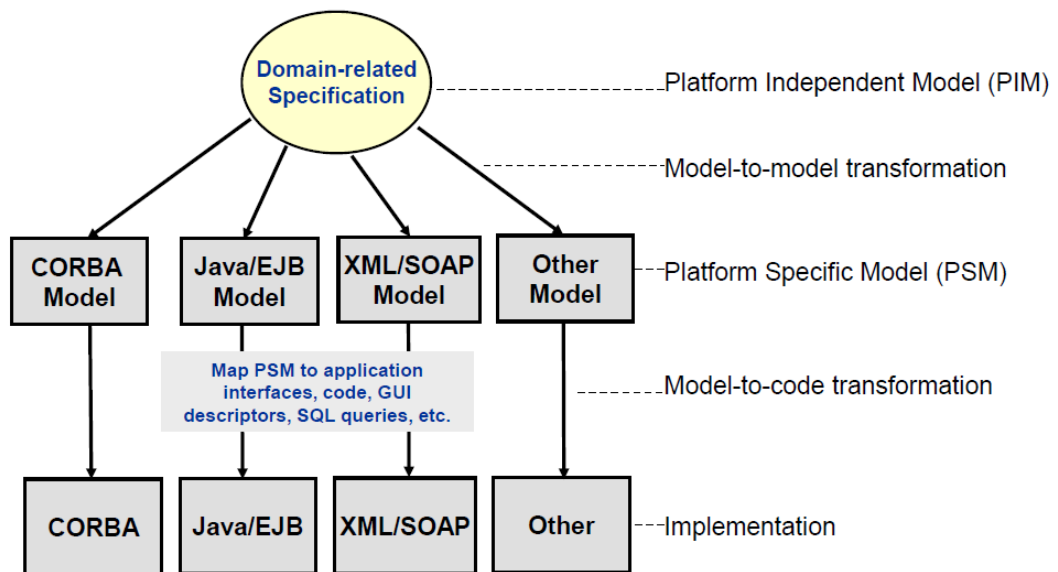


Figure 2.1. MDA models and transformations [Ley13]

of the aforesaid the information of the original and generated (transformed) models is the same [OMG14].

Transformations are supported via mappings and patterns, which specifies how the models of the MDA have to be transformed i.e. what is the correspondence between elements of the models and how the transformation has to be done.

2.2 Model Difference

Author: Michael Zimmermann

A large number of algorithms for comparing model versions have been proposed. One class of them is based on logging. The logging-based approaches (as e.g., [SZN04], [LO92], [HK10], [Kög08]) log user edit operations and store them with the corresponding model. A big problem of these approaches is the dependence on closed development environments. Of course changes in the model are just logged if they were made in the monitored editor or tool. In addition, only direct revisions of a model can be compared. Thus, transformed models or imported models can't be compared using this logging-based approaches.

The remaining model comparing approaches can roughly be divided into (a) algorithms that can only detect elementary low-level model changes and (b) algorithms supporting the semantic lifting of model differences.

There are many approaches belonging to group (a). For example [The14b], [Tae+14], [BP08], [Bra11] and [Sof14a]. However, the problem with these approaches is the difficulty

to understand the results of the algorithms or tools. This is because even a small user modification of a model can result in a lot of low-level technical changes that are hardly to understand by the user.

The approaches of group (b) address this problem. Here the detected low-level changes are grouped into user edit steps and thus lifted on a higher abstraction level. [MB+14] provides a UML-specific EMFCompare extension that enables the grouping of low-level changes into high-level change operations. But for each edit operation the code that detects this high-level change operations must be implemented manually. Moreover, this extension only is suitable for UML models.

Another approach from Könemann [Kön10] focuses on model patching. Nevertheless, he addresses the problem of grouping atomic changes into "abstracted semantic changes" that "are closer to the user's intention". The process he describes uses different strategies like name patterns or OCL queries. But as part of this process the user needs to interact e. g., check the correctness of introduced abstractions.

A much more automated approach is SiLift [Keh+14a] (see chap. 5.1.6). Here, in contrast to Könemanns approach, the recognition rules which are needed for the semantic lifting can be derived from the meta-model specification. Also, the execution of these rules and thus the semantic lifting can be fully automated.

2.3 Model Transformation

Author: Rene Trefft

Model Transformation is an essential activity in Model Driven Development (MDD). Models are modified, translated to intermediate models and finally code is generated. Due to this importance, different Model Transformation approaches have been developed. In the following a closer look on in-place transformation approaches for models based on the Eclipse Modeling Framework (EMF) is given. EMF is a well-known and widely used modeling framework which provides code generation facilities for creating applications based on structured data models. [Are+10]

EMF Tiger [Bie+06] is a Model Transformation approach which provides a purely rule-based transformation language that supports simple attribute changes only. Just negative patterns as application conditions are accepted.

Kermeta [Ker] is a textual approach that supports behaviour definition through an imperative, objected-oriented action language.

The Epsilon Wizard Language (EWL) [Kol+07] is suitable for small in-place transformations. A so-called wizard can be compared with a rule. It consists of a guard, a title and a do-section where the transformation is imperative and object-oriented programmed. The effects of a transformation can be undone or redone.

In Moment2 [Bor07] the transformation is realized by rewriting logic based on Maude. A rewriting theory can include complex conditions as OCL constraints [Ocl]. However, such rewrites can't be grouped to larger transformation modules.

2. Related Work

Transformations in MOLA [Mol] are based on MOLA diagrams. A MOLA diagram consists of pattern-based rules, control constructs like loops and sequences, subprogram calls and further graphical statements. MOLA doesn't support amalgamation or a similar concept for the application of transformation rules in parallel. Fujuba's [GHS09] story diagrams provides similar language constructs as MOLA, but also don't offer amalgamation. Moreover, MOLA as well as Fujuba don't provide a formal basis for further model transformation validations.

The transformation language Viatra [BV06] which combines graph transformation with abstract state machine (ASM) concepts is also rule- and pattern-based. In Viatra, modeling languages must be specified by a proprietary meta modeling approach. Models in standard meta modeling formats like EMF can be imported. The language provides advanced features, e. g., recursive graph patterns, generic and meta-transformations and ASM control structures.

Henshin is the only Model Transformation approach which directly operates on EMF models. It provides a comprehensive and expressive graphical transformation language, supports endogenous as well as exogenous transformations and is completely based on algebraic graph transformation [Ehr+06; BET10; Kus00].

GROOVE [KR06] is also graph transformation based and supports advanced concepts like nested application conditions and amalgamation. Indeed, regular expressions can be used for matching which Henshin doesn't support. However, model checking is realized by Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) formulas which are less expressive than the μ -calculus formulas supported by Henshin through the CADP model checker [Gar+07]. Last but not least, GROOVE doesn't support EMF models yet.

Project Management

The central part of the Co Wolf project was a time-boxed development phase of 8 weeks. With a given development team size of 12 persons and a 40 hour week, this sums up to 480 hours of development per week and about 3800 hours in total. Obviously, it is essential to choose a development process and infrastructure that meets the requirements of managing the short timespan on the one hand, the relatively big team and therefore highly parallel working method on the other hand.

In this chapter, we describe Scrum as foundation of our development process in Section 3.1, the team structure and roles (Sections 3.2 and 3.3) as well as the idea of Continuous Delivery and tool infrastructure we used to support the development process (Section 3.4).

3.1 Scrum

Author: Philipp Niethammer

Scrum is an agile software development process framework conceived by Ken Schwaber and Jeff Sutherland, published in 1997 [SS14; Sch97]. It is based on the realization that, unlike other process models assume, e.g. the Waterfall model [Roy70], often the development process cannot be planned completely in the beginning.

Scrum instead splits the development in time-boxed phases, Sprints, usually of one month length or less. The goal of each Sprint is a usable, stable product increment. Unlike in iterative development where each iteration's goal is defined in the project planning, the development decides on the content of a Sprint at its beginning. The decision is supported by the Product Backlog, a prioritized list of requirements, so-called User Stories. This list is continuously updated to match the needs of stakeholders.

Consequently, Scrum results in a flexible, agile development process that can deal with unclear a priori requirements as well as react quickly to changes in the stakeholders' interests.

When we began the project, there was only a vague goal and it was clear, that the customer will itemize his requirements and priorities during the development phase. Thus, we decided to use Scrum and as it is not a fixed process [SS13], adapted it to our needs.

We organized the development process in sprints of a week length each. This sounds quite short at first glance but considering the restricted development time of 8 weeks and therefore a total of 8 Sprints it was essential to obtain the wanted flexibility. Addition-

3. Project Management

ally, with the 480 hours of development time per week stated above, it was just within manageable limits.

3.1.1 Sprint Schedule

Each Sprint started with the Sprint Planning. In this meeting, we discussed what items of the Product Backlog can be done during the week and specified the technical tasks that are necessary to complete these items. After the goal of the Sprint was set, potential dependencies and problems were located and the tasks were assigned to the different members of the team.

At the end of a Sprint, we used the Sprint Retrospective and the Sprint Review [SS13] to reflect the sprint. First, in the Sprint Retrospective we explicitly inspected the last week in regard to issues in the process and the communication in the team and with external persons. After that, we had a look at the goals of the Sprint and assessed whether these goals have been reached or not. In the latter case, we discussed the problems that lead to the delay and reviewed the consequences.

These framing meetings were supplemented by Daily Scums, a short daily 15-minute event where each team member explained what he has done and what he is going to do the next day. This meeting mainly helps to synchronize activities and solve arising problems in an early phase.

3.1.2 Tasks Management

The Backlog items normally are User Stories, a non-technical description of a requirement from a stakeholder's viewpoint as described in Chapter 4. However, during development certain tasks arise that cannot be intuitively be formulated as user story. These are mainly bug fixes, code refactoring and usability improvements. To manage these issues in the Scrum infrastructure, we maintained them separately and included them in our Sprint Planning. Additionally, we allowed to add issues of this kind during the Sprint, as well. This was the case if bugs or enhancements were wrongly regarded as unimportant during Sprint Planning but become necessary for the ongoing development. The dynamic addition to the Sprint also worked as a buffer if a developer had to wait until dependencies were resolved or had already finished his tasks.

In total, we worked on up to 50 tasks per Sprint. Although there is lots of tool support for managing tasks in an agile environment, they were mostly considered as unclear and not optimal to gain an overview of the process. Hence, we decided to use a simple ticket system (see Section 3.5.2) and additionally keep track of all tickets and their process on a physical board. According the suggestion of Sutherland, Downey, and Granvik [SDG09], we separated it into the columns Product Backlog, Sprint Backlog, In Work and Done and added the column Pull Request to satisfy this special state in our revision system (cf. Section 3.5.2).

3.2 Team

Author: Jonas Scheurich

The following table documents the team experience in the different technologies of the CoWolf project.

Table 3.1. Team experience

	Christian Karl Bernasko	Manuel Borja	Verena Käfer	David Krauss	Michael Müller	Philipp Niethammer	Tim Sanwald	Jonas Schetrich	David Steinhart	Rene Trefft	Johannes Wolf	Michael Zimmermann
Plugin Prototype	•					•			•			○
GUI	○		•		○	•		○	○			○
Meta Models	○	○	•		○	•	•	○			•	
Graphical Editor			•								•	
Model Versioning					•							
Evolution				•	○	○	•	○		○		•
Co-Evolution Components → Fault Tree	○	○		•								
Co-Evolution State Chart ↔ DTMC					•				○			○
Co-Evolution CTMC ↔ DTMC			○			•	•					
Co-Evolution Sequence Diagram → LQN							•					
Co-Evolution Fault Tree → CTMC				○	•		•					
Co-Evolution Framework		○		•	•	•						•
Performance Analysis		•				○		•				
Safety Analysis		•										
Reliability Analysis						○	○	•				
Continuous Delivery	•								•			
Automatic Tests	•											

Main experience in this technology | •
Some experience in this technology | ○

3.3 Roles

Author: Verena Käfer

We mainly adopted the Scrum roles but also added some team specific roles:

3. Project Management

▷ **Project leader:** Philipp Niethammer

The project leader was responsible for the organization of the whole project. He planned the meetings with the supervisor and the customer, structured the daily scrum meetings and organized the sprint plannings.

▷ **Product Owner:** Tim Sanwald

The product owner was responsible for the backlog. He organized the tickets and the prioritization of them.

▷ **Sys-Admin:** Christian Karl Bernasko

The Sys-Admin was responsible for the technical administration of the project. He organized the repositories and the Maven project structure.

▷ **Quality Manager:** Rene Trefft

The quality manager was mainly responsible for the correct naming and categorizing of all plug-ins to have a consistent naming and no unnecessary dependencies.

▷ **Documentation Manager:** Jonas Scheurich

The documentation manager was responsible for the final report. He distributed the chapters and developed the L^AT_EX infrastructure for the final report.

▷ **Developer:** Everybody

▷ **Typing Master:** Verena Käfer

The typing master was responsible to protocol every customer meeting and send a protocol to all participants.

▷ **Fun Master:** David Steinhart

The fun master was responsible for the team spirit. He organized activities for the team so we could get to know us better and relax after work.

3.4 Continuous Delivery

Author: Christian Karl Bernasko

Many companies now using agile development strategies to produce software in a better quality. The agile development practices focusing on customer involvement and fast feedback for developers. The most companies using continuous integration to enhance their workflow. Continuous integration fits well in these agile paradigms. However, when it comes to delivering software to the user, then there is less knowledge provided from these agile methodologies. The methodology continuous delivery covers the need to deliver software changes to the customer and also focuses on agile practices. As a team we decided that we want to use the Scrum software development framework. The

3.4. Continuous Delivery

main part of Scrum are sprints. During a one week sprint many software changes were introduced to the CoWolf project. To maintain these software changes we decided at the beginning of the project to use the continuous delivery methodology. Continuous delivery was introduced by Jez Humble and David Farley [HF10]. Continuous delivery has two main aspects, collaboratively implementing source code (continuous integration) and delivering the software changes to the customer. To have continuous integration and delivery to work hand in hand we needed to implement a continuous delivery pipeline. A full featured continuous delivery pipeline consists of the following elements: a commit stage, an automated acceptance testing stage, an automated capacity testing stage and a manual testing stage. In this section we explain, how we implemented the continuous delivery pipeline, we oriented us on the described purpose from the seminar paper [CKB14] to fit the need of the build pipeline for the Co Wolf project.

3.4.1 Continuous Integration Process

Author: Christian Karl Bernasko

For developing source code as a team, we focused on the process that we adapted from the article “Continuous Integration on a Dollar a Day” by James Shore (27th February, 2006). Initially all members of our team agreed to one rule, “From now on, our code in the master branch at our Git repository will always build successfully”. We applied the rule by performing the following steps. Every team member rebases its source code several times a day. This means all team members have an up-to-date branch. Before committing new source code into the version control system, the developer needed to find out if anybody has the CoWolf. The developers use the CoWolf as a token. The CoWolf is a action figure that looks like a wolf Listing 3.1. Only the developer who has the CoWolf is allowed to interact with the master branch in the version control system. This means committing new code into the master branch. First we run the build and the test scripts locally on our machine. If the build succeeded, then the developer was allowed to merge the local branch with the master branch. After the developer merged the local development branch into the local master branch, he executed the build again. When the build passes, he committed the local master branch into the remote master. Instead of using a bell like James Shore used one, we used a traffic light which can be seen in Listing 3.1. The traffic light indicates the last committer and the build status. Over the development time we learned that a merge token and a traffic light are important tools for using the continuous integration methodology successfully. The merge token helped us to commit without creating merge conflicts. Whereas the traffic light together with an indicator of last committer made the continuous integration server and build status more physically available.

3. Project Management



Figure 3.1. CoWolf merge token and traffic light

3.4.2 Continuous Delivery Infrastructure

Author: Christian Karl Bernasko

To develop the CoWolf software we used several processes and tools. Discovering the best tools, and learning how to use them effectively, was a part of this project. The CoWolf project is an Eclipse based plugin, prior to this technology we needed to have a Build tool that supports Eclipse based applications. We choose the Apache Maven Project as our build tool, which is also a project management and comprehension tool. Maven allowed us together with Tycho to build Eclipse plugins (see Section 3.5.3). The main goal of the project was to build CoWolf collaboratively, to meet this goal we used a continuous integration server called Jenkins (Section 3.4.3). Jenkins also stored our compiled binaries into a artefact repository (Section 3.4.4), which allows other developers to retrieve the binaries of CoWolf without recompiling them. To get an overview of our product metrics we used Sonarqube to analyse our source code and binaries (Section 3.4.5). The Sonarqube analyse was trigger by Jenkins. To manage the collaboratively development we needed a version control system. We used Git as version control system together with the Github ecosystem (see Section 3.5.2). All used tools are described in the following sections Listing 3.2 .

3.4.3 Continuous Delivery with Jenkins

Author: Christian Karl Bernasko

Continuous integration server are also called build servers. Build servers are the center of the delivery pipeline. They provide automatism to build software and provide fast feedback with reports. We could choose a build server from a variety of continuous integration servers implementations. We used the most popular continuous integration servers called Jenkins. Due to Jenkins expandability with plugins, we implemented a delivery pipeline with several plugins.They are listed below.

- ▷ Git plugin
- ▷ Build pipeline plugin

3.4. Continuous Delivery

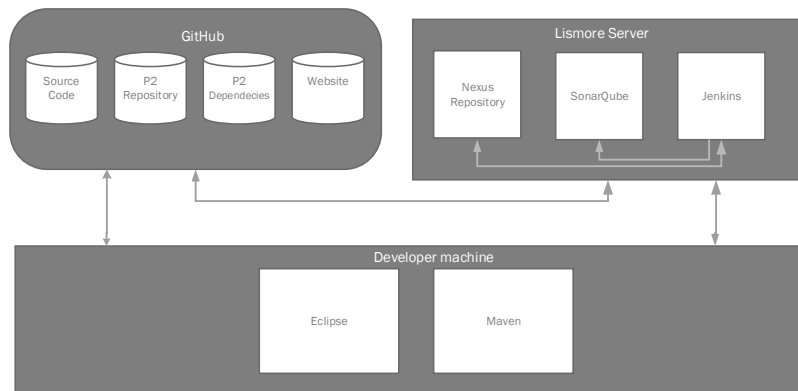


Figure 3.2. Continuous delivery infrastructure

- ▷ SonarQube plugin
- ▷ Maven plugin

Jenkins, which is initially a scheduler used for cron jobs, can be expanded with the use of the Git plugin to listen to a commit. We needed to configure Jenkins in order that every time when a member pushes new code into the Git repository on Github, Jenkins gets a notification and schedules a new build job to compile our source code. Github supports external build server by providing a service which allows to specify a URL that gets called every time a change happens in the Git repository.

To use the continuous delivery approaches we needed to implement a delivery pipeline. With the build pipeline plugin it is possible to build a view that looks like a pipeline. This makes it possible to overview every stage of the delivery pipeline at one place. In Jenkins we can specify different build steps, the build pipeline aggregates the build steps and visualize them as a pipeline. To implement a visualized pipeline we need to specify the following build steps:

- ▷ build stage
- ▷ analyze stage
- ▷ deploy stage

An important advantage of the build pipeline plugin is that we can concatenate several build jobs to a pipeline. If the build job succeeds, Jenkins schedules the next pipeline stage. We used a small subset of the delivery pipeline that fitted our needs. We used a build stage, where we used the Maven build tool to compile and verify our product. The analysis stage where we analyzed our product and the deploy stage, where we deployed the binaries as p2 update site into a separate Git repository.

3. Project Management

The Build Stage

In the build stage of the delivery pipeline, we had to do several tasks such as compiling, code analysis, and executing unit tests or storing the build output into the artifact repository. Jenkins supports these tasks by specifying pre and post as well as up and downstream actions. With these actions it is possible to define which tasks should be executed before or after the other tasks. For example, we can specify a post action to commit the binaries to the artifact repository after compiling successfully the source code. The build stage compiles the source code of our product. For this part we used the build tool Maven, the advantage of Maven over ant for our project was that Maven integrates with Tycho, which we used to build Eclipse specific plugins. In the build stage Maven resolves all our dependencies and then compiles our source code. If Maven finished the compilation process it stored the resulting binaries in our Nexus artifact repository, this allowed us to reuse the binaries in the Analyze phase. During development of the CoWolf project, we enhanced the build step by adding a notification script. The notification script allowed us to get notified via Skype if a build step failed or succeeded. If the build process passed successfully we trigger the next stage the analyzing stage to start.

The Analyze Stage

We used SonarQube in the analyze stage to analyze the software metrics of our product. At first the analyze stage was part of the build stage, but due to the long compilation and analyze time we were forced to split the build phase in two parts. We needed to make this decision to be compliant with the continuous integration methodology. We described in the previous section that we agreed as a team that as long as a team member is in possession of the merge token, nobody is allowed to introduce new code changes into the master branch. The high frequency of code changes which lead to many commits and every commit to a new scheduled build which took an average build time of ~20 minutes, while the analyze part and the artifact deployment part also took about ~15 minutes we needed to split the build phase into a build phase and analyze phase.

Deploy Stage

The last stage of the delivery pipeline is the release stage. In the release stage, we deployed our binaries into a Git repository on Github. A great feature of Eclipse is that plugins can be shared via a p2 repository also called update repository. Maven together with Tycho, created for us at the build stage, the p2 repository. We only needed to upload the repository automatically to a public place where the end user can download it. Since we made in this project heavily usage of the Github ecosystem (Issues, Git, Wiki, Website), we searched for an idea to also place the p2 repository on Github. We used a custom script to commit our binaries into a separate Git repository called p2_update_site. A feature of Github is to view files in a repository in a raw view, we used this feature to allow an

3.4. Continuous Delivery

Eclipse instance to download the p2 repository. The address to download this repository is https://github.com/DevProjectSS2014/p2_update_site/raw/master. This link is only usable within the Eclipse installation dialog. After a new version was deployed to the p2 repository, we were able to install it into Eclipse. This helped us to get early feedback about missing dependencies. The missing dependencies could not be detected during the build and analyses stage nor on the developer machine. Common failures that we detected after deployment were missing third party dependencies. The dependencies were missing in the feature bundles (needed to be added manually separately) and also project options which weren't enabled. This led to the situation that after installing the software, icons and functionalities were missing.

3.4.4 Nexus Pro Repository Manager

Author: Christian Karl Bernasko

Nexus is a version control system equally to Git, but we used Nexus to store artifacts instead of source code. Software artifacts we understand to be the components of the software like executable code, configurations, and database data. When we use Java, the artifacts can be jars, wars, ears and fully developed libraries or collections of libraries. Software projects are rarely developed without dependencies. For instance for our projects we used the third party libraries Sidiff (Section 5.1.5) for model comparison and Logback for logging. These dependencies have to be administered, which requires more effort if the project contains many dependencies. When projects are developed interoperable they usually need different versions of the same dependencies. If we consider an antipattern for administering dependencies we would use a shared folder, which contains the dependencies and a list where all the needed dependencies are listed. The developers would add the needed dependencies manually to their project. The problem is that the list of the needed third party dependencies and the shared folder can be disharmonious. We agreed as a team that this is not a solution for our project, because it makes the development project setup more complex. However, to manage dependencies comfortably we can use an artifact repository. We used the artifact repository Nexus together with our build tool Maven. Maven has an integrated package manager. Developers can check in developed libraries in the artifact repository to share the libraries with their team members. The build tool then uses the package managers to check out libraries from the artifact repository to resolve dependencies and build the source code with the latest libraries. Another benefit, which is needed for the delivery pipeline, is that it is possible to store a successfully compiled build into the artifact repository. Jenkins is then able to check out exactly the same build in another build job, without recompiling the source code. During development we learned that the part of resolving dependencies is time consuming with the build tool Maven. The Nexus repository manager has the possibility to aggregate several third party p2 repositories to one repository. The benefit to proxy the third party p2 repositories is, that we are independent from the latency. However, we could not use this feature due to the limited server resources.

3. Project Management

3.4.5 SonarQube

Author: Christian Karl Bernasko

SonarQube is an open platform to manage code quality. The framework includes a wide range of quality analysis technics and provides them for many programming languages. We are using SonarQube to verify that we develop software with a high quality. For this purpose we are using several quality metrics e.g. comments, code duplication, code complexity, coding standards, code coverage, potential bugs, design and architecture detection and technical depth. SonarQube aggregates the different metrics and visualizes them. The SonarQube system consists of two parts, the data collector SonarQube runner and the SonarQube server. The SonarQube runner needs to be installed into the project root. Then he collects all specified metrics from the project and publishes them to the SonarQube server. The SonarQube server visualizes all data thru its web interface. The SonarQube plugin for Jenkins connects Jenkins with the SonarQube sever. This makes it possible, to integrate the metric collection process into the build pipeline. Jenkins allows us to specify a pre-build process, which executes the SonarQube Runner to collect all data. SonarQube has the possibility to define a quality goal, for instance the unit test coverage must be higher than 80 %. If the metric exceed the predefine threshold, then Jenkins aborts the job and response the issue. With this process we can verify that we only have source code in a good quality in our repository. We learned during development that CoWolf heavily relies on generated code. This lead to the situation that the software analyses showed metrics of third party code rather developed code by our development team. An additional issue was that our source code was mixed up together with the generated code, because of this situation we were not able to exclude only third party generated code from the analyses.

3.5 Tooling

To get the maximum out of the project we used several tools to support us. First of all we used Eclipse as environment for the development. As a version control system we decided to use Git with GitHub as host for the project. To have a up-to-date overview over the latest build and a continuous delivery we used Maven. All used tools are described in the following sections.

3.5.1 Eclipse

Author: Rene Trefft

We used Eclipse Luna (4.4.0) for the development of CoWolf, because a requirement was to develop CoWolf as plug-in for the Eclipse Platform. Further every team member had already a strong experience with this IDE.

Eclipse Plug-ins

The architecture of the Eclipse Platform is extensible, so every developer can contribute new components, called plug-ins. Often a plug-in is also called OSGi bundle, because Eclipse is based on Equinox, the reference implementation of OSGi which is a specification of a component and service framework for Java. OSGi specifies a bundle as a unit of modularization. In the following we will use the Eclipse term plug-in. Keep in mind that both terms are (almost) interchangeable. [Vog13b; Vog14]

A plug-in is a Java project with additional meta data. These meta data are stored in the files META-INF/MANIFEST.MF and plugin.xml relative to the root of the plug-in. [Vog13b; Vog14]

The manifest file is a key-value-based file and especially specifies the Bundle Symbolic Name which is the unique identifier of the plug-in. Further it explicitly defines which packages of other plug-ins (attribute Import-Package) or complete plug-ins (attribute Require-Bundle) are used and which packages will be exported (attribute Export-Package). In this way the dependencies and the interface of the plug-in are effectively controlled. Listing 3.1 shows a manifest file as an example. [Vog13b]

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: CoWolf Example
Bundle-SymbolicName: de.uni_stuttgart.iste.cowolf.example;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: University of Stuttgart Institute of Software Technology
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: de.uni_stuttgart.iste.cowolf.model,
    de.uni_stuttgart.iste.cowolf.model.commonBase,
Export-Package: de.uni_stuttgart.iste.cowolf.example,
    de.uni_stuttgart.iste.cowolf.example.util
Require-Bundle: org.eclipse.core.runtime;bundle-version="3.10.0",
    org.eclipse.core.resources
```

Listing 3.1. Example of a plug-in manifest.

The plug-in file is an Eclipse-specific file and therefore not specified in OSGi. It defines extensions and extension points. An extension point can be seen as a socket whereas the extension is a plug that connects to it. If a plug-in wants to allow other plug-ins to extend or customize its functionality it declares an extension point. It consists of an ID, a human-readable name and an XML schema document that represents the structure to which extensions must conform to. An extension consists of an ID and the XML node that complies with the schema of the extension point. The extension registry contains the declared extension points and extensions and provides an API to retrieve all installed extensions of a certain extension point. A key thing to notice is that no Java classes have been loaded after the extensions are retrieved, but only XML data, so the time for class

3. Project Management

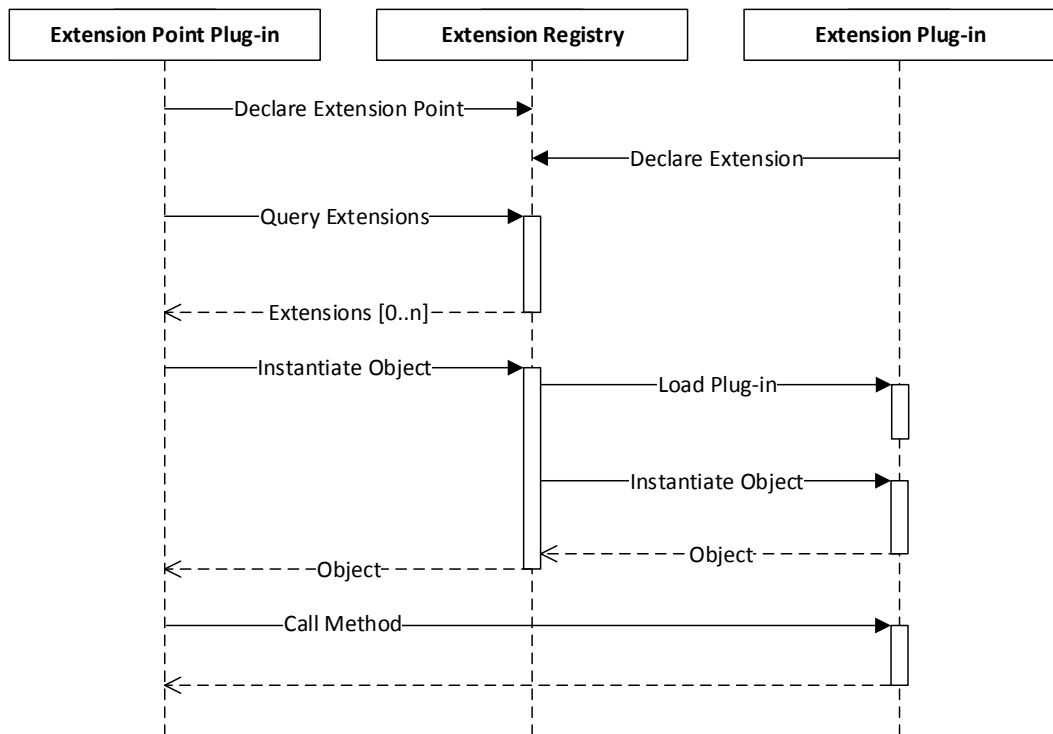


Figure 3.3. UML sequence diagram showing the interaction between Extension Point Plug-in, Registry and Extension Plug-in on a method call. [Bar07]

loading is saved. If an extension point plug-in defines a Java class name in its schema and finally needs the class it asks the registry for loading and instantiation. Thus, it's up to the plug-in when classes of the extensions should be loaded. Listing 3.3 shows the interaction that occurs if an extension point plug-in wants to call a method of an extension plug-in. [Bar07]

The Eclipse Platform already provides a lot of extension points. For CoWolf we mainly supply extensions for Eclipse extension points to customize and extend the Eclipse UI, e. g., add the CoWolf perspective and wizards.

Eclipse Fragments

A fragment is an optional extension of a plug-in which is called host plug-in. It has also a manifest which can contain the same attributes like a plug-in manifest and the host plug-in. At runtime the fragment is logically merged with its host and both are represented as one plug-in.

Eclipse Features

A feature bundles plug-ins or other features and can be installed in the Eclipse Platform. It can specify dependencies which must be present before installing. Further it can include a description, copyright and licensing information. Also, a branding plug-in can be specified that extends the About dialog of Eclipse. All mentioned data are defined in a file `feature.xml` in the root of the feature.

Eclipse Target Platform

The target platform consists of a set of plug-ins which can be used (imported) in your plug-in and fragment projects. By default the Eclipse installation is used as target platform. It can be edited in the Eclipse preferences. The target platform can be, for example, extended with the plug-ins contained in a P2 repository or directory.

Equinox P2

The Eclipse Platform provides an installation and update mechanism called Equinox P2 [Vog13a]. A P2 repository, also called update site, consists of plug-ins and features that can be grouped into categories. In Eclipse a P2 repository can be generated with an Update Site Project. It contains a `category.xml` file which defines the categories and contained features and plug-ins.

Eclipse Product

An Eclipse product is a standalone version of an Eclipse Platform based tool. It can be created for different environments. To create a product you must first create a product configuration project. The product file in the root of the project contains all information about the product. For example each product has an ID, branding stuff like icons, a splash screen and text and the plug-ins respectively features (dependencies) the product consists of. You can create a plug-in based or feature based product. Recommended is the latter approach, because otherwise your feature information like the description or licensing information are not included in the product. Especially a definition of a product must be created and defined in the product file. A product consists of an ID and a plug-in that provides an extension for `org.eclipse.core.runtimes`. This plug-in further includes the splash screen and icons defined in the product file and must be part of the product. Additionally, an application must be specified which represents the default entry point for the product. A new application can be created or an existing one reused. In case of a tool which should be based on the Eclipse IDE you can simply choose `org.eclipse.ui.ide.workbench`.

3. Project Management

Eclipse PDE

The Plug-in Development Environment (PDE) of Eclipse provides tools for the development, testing and debugging of Eclipse plug-ins, fragments, features, update sites and products. For example it provides different projects and form-based manifests editors, e. g., a Feature Manifest or Product Configuration editor. Equinox can be directly launched with a defined set of plug-ins and fragments (or features) from your workspace and the target platform.

3.5.2 GitHub

Author: Tim Sanwald

GitHub is a web-based hosting service for software projects which offers a distributed Version Control System (VCS) and several collaboration features like wikis and task management. GitHub is one of the most popular services for hosting open-source projects, probably through the easy to use graphical interface and in contrast to other open-source hosting services, the main part at GitHub are the users and not the projects or repositories.

As VCS git is used at GitHub, which is a distributed system with emphasis on speed, data integrity and non-linear workflows, more details on how we used git is described in the following section. Afterwards, our use of the issue management system is explained.

Git Workflow

Git is a VCS like SVN or Mercurial and for this project wouldn't be a problem to use any of this, but we decided to use git in the beginning of the project. This decision was based on three facts, first most of us had worked with git before. Second git is more error save than SVN or Mercurial. At last the project should be easily published at the end of the project, thus, we decided to use GitHub as provider and therefore we had to use git as VCS.

Our Git workflow is based on [Cha01] and is defined for each user story:

1. Create a new branch named with the id of the user story and a short description. This branch should be based on the current master commit.
2. Work on that branch locally.
3. Commit that branch regularly to the remote repository to inform others about the progress.
4. When you think the user story is fulfilled create a Pull Request on GitHub.
5. Another person is reviewing the pull request and is commenting parts which have to be improved or changed.
6. If nothing has to be changed, the person can merge it into local master and check if everything is working, too.

7. If everything works, it can be pushed to the remote repository and Jenkins is automatically called to build it.

Issue Management

Issues are one of the key feature on GitHub, they enable users to manage tasks which have to be done. Also they can use them as a bug tracking system. An example of an issue is given in Figure 3.4. Each issue consists of a title, a milestone, an assignee, labels and comments:

Title describes the topic of the issue.

Comments can be created by each user. The first comment can be interpreted as a detailed description of the issue. Followings are used to specify the issue or discuss with the author, this is the case when problems have to be found or proofed if they exist on several systems.

Milestone describes a phase of the project.

Assignee refers to the contributor who works on this issue which is recommended.

Labels can be defined by the user to signify priority, category, or any other information that could be useful. In our case we used this to specify the type of the issue, so whether it is an user story, task, bug or an enhancement.

We used a combination of online issues and offline post-its. Thus, we printed out the issues and put it onto a scrum board to better visualize the progress of the current sprint and our overall process.

3.5.3 Maven

Author: Rene Trefft

For the automation of the building process of CoWolf we decided to use the Build-Management-Tool Apache Maven which can be used for many building tasks like compiling, testing, packing and the deployment of software. Moreover Ant scripts can be executed for arbitrary tasks. All functionalities are provided by plug-ins, so Maven is completely extensible. A plug-in consists of goals that can be executed. For example the Surefire plugin provides the goal test to execute test cases. Goals of plug-ins are bound to phases. A list of phases constitute a Maven lifecycle. If we for example pass the phase package to Maven all goals associated with all phases until package will be executed. [The14a]

The project configuration is stored in the XML file `pom.xml` (Project Object Model) in the root of the project. Each Maven project, generally called an artifact, is identified by a `groupId` that represents the organization, an `artifactId` and a version. The qualifier `SNAPSHOT` in a version, e. g., `1.0-SNAPSHOT`, stands for a not yet released artifact. Further

3. Project Management

Sirius: Copy/Delete/Move representation when done to model. #377 Edit New Issue

Closed philippniethammer opened this issue 19 days ago · 9 comments

philippniethammer commented 19 days ago

When a model file is copied/deleted/moved, the representation file must be updated to match.

follows [#386](#)
refs [#372](#)

philippniethammer added the **type:bug** label 19 days ago

philippniethammer added this to the **Sprint 9** milestone 19 days ago

verenya self-assigned this 19 days ago

timsanwald added the **printed** label 18 days ago

verenya commented 16 days ago

It is not possible to update the representation and keep the layout

- First approach: copy the aird file and manually update the target of the diagram.
Problem: The copy function disguises as an add and this causes the aird file to delete all existing layouts. The change of the target does not change or prevent this.
- Second approach: Create a new aird file and use a MoveRepresentationComand to move the representation to the new aird file
Problem: The session of the old aird files does not find any representations any more. As the matching resource does not exist any more, the result of the `getRepresentations()` is always empty. And it's not possible to move nothing :-)

Possibilities:

- Rename and move the aird file with the resource and lose the layout
- Just reset the target. Problem with this: It only works one time, afterwards we cannot find the matching aird file anymore, because it is named differently.

Labels

- printed
- type:bug**

Milestone

- Sprint 9

Assignee

- verenya

Notifications

- Unsubscribe

You're receiving notifications because you commented.

3 participants

- philippniethammer
- verenya
- timsanwald

Lock issue

Figure 3.4. Example for an GitHub issue.

each artifact must define a packaging which defines the type of the project. For example the packaging jar is supported by Maven out of the box. Further types of packaging are provided by Maven plug-ins, so they must be specified in the POM first. The packaging defines the phase-to-goals-bindings. [The14a]

Maven plug-ins are located in external Maven repositories and will be downloaded to the local Maven repository. Normally any dependencies defined in the POM are also fetched from Maven repositories. In our case the Maven plug-in Tycho (see next section) was used for building Eclipse plug-ins. Tycho needs dependencies from a P2 repository.

Associations between Maven projects can be defined by POM-inheritance and -aggregation. A project can define a parent project in its POM to reduce the redundancy and thus the maintenance effort and probability for faults. Furthermore a project with packaging pom can define sub modules (projects). A phase executed on such a project will be executed on all sub modules in the order they are specified in its POM. [The14a]

Plug-in Building

Maven doesn't provide packaging types for e. g., Eclipse plug-ins or features out of the box. Therefore an appropriate plug-in must be added to the POM of the desired project. Currently there exist two approaches for Eclipse plug-in building with Maven.

The Maven bundle plug-in is the POM-first approach which means your dependencies are defined in the POM and are fetched from a Maven repository. It's the standard Maven way of dependency management. Also attributes like the packages to export will be defined in the POM. At build time the bundle manifest will be generated. The main disadvantage of this approach is that you have a standard Java project (with Maven project structure) instead of a plug-in project and thus no Eclipse PDE support. Especially for the development of an Eclipse Platform based tool this is an in-acceptable point.

Therefore we decided to use the Manifest-first approach, the Maven plug-in Tycho. It uses the metadata of Eclipse plug-ins, e. g., the bundle manifest for building, so you have to create an Eclipse PDE project which means all PDE tools are available. Tycho supports plug-ins, fragments, features, update sites, products and also RCP applications which are not relevant for us. Thus, the appropriate packaging must be defined in the POM. An important thing to notice is, if Tycho is added to your POM, any dependencies declared there will be ignored. Instead, you have to specify P2 repositories which contain the necessary plug-ins, because Tycho internally uses parts of Equinox P2 for dependency resolution.

Maven Integration for Eclipse

The Maven Integration for Eclipse, also called m2e, is an Eclipse plug-in that provides support for Maven projects in Eclipse. Maven builds can be directly launched from Eclipse by using the supplied, embedded Maven or an already installed, external Maven version. Several wizards are provided, e. g., for the creation of Maven projects and a POM editor.

3. Project Management

Since Eclipse Juno m2e is supplied with several editions of Eclipse, e. g., the Java Developers edition.

We use m2e for managing our Maven projects (especially their POM files) and launching a local build of CoWolf within Eclipse.

Maven Project Structure

The parent project `de.uni_stuttgart.iste.cowolf.parent` aggregates all CoWolf projects which should be part of the Maven build process. Thus, it declares these projects as modules in its POM and has the packaging `pom`. We've defined as group ID `de.uni-stuttgart.iste.cowolf`, as artifact ID the project name (Maven convention) and as version `1.0.0-SNAPSHOT`. Further all necessary P2 repositories were added, so that all plug-ins of our target platform are also available during the build respectively can be resolved. Moreover we added necessary Maven plug-ins, especially Tycho.

The aggregated projects are plug-ins, fragments, features, update site projects and a product configuration project containing the product file. They're all define the parent project as parent, so they inherit all elements except the artifact ID and the packaging. The artifact ID always begins with `de.uni_stuttgart.iste.cowolf`, the packaging depends on the project type, e. g., the Tycho packaging `eclipse-plugin` was used for a plug-in.

Every test project of CoWolf, e. g., `de.uni_stuttgart.iste.cowolf.model.statechart.tests` was created as fragment, because in this way classes which are only visible within its package can also be accessed and thus tested.

Features were created for sets of CoWolf plug-ins which belong together, e. g., the CoWolf Core (`de.uni_stuttgart.iste.cowolf.core.feature`) or the Statechart Evolution. These features with their contained plug-ins were added to the update site project `de.uni_stuttgart.iste.cowolf.p2update`, more precisely said their category file. The P2 repository is generated during build and stored in a separate Git repository on GitHub. It's used for installing or updating CoWolf. Another update site project `de.uni-stuttgart.iste.cowolf.p2dependencies` was created for necessary plug-in dependencies which are not available in an external P2 repository. This applies to the plug-ins of the logging framework we use (see chap. 5.3). We decided to separate this project from the standard build process, because changes occur rarely. It is also stored on GitHub.

To deliver CoWolf also as a standalone software we created an Eclipse product. The product configuration project is `de.uni_stuttgart.iste.cowolf.product` and the product project `de.uni_stuttgart.iste.cowolf`.

Requirements

At the beginning of the development the most of us had limited knowledge about models, co-evolution and of the tools we have to use. In order to understand what was the expected result of the project, we had meetings between the whole team and the advisors. During these meetings we identified requirements which our product had to fulfill. The result of this requirement analysis is described in the Chapters 4.2 to 4.8. In order to establish the input information, we included the requirements as user stories.

An user story represents the 'who', 'what' and 'why' of a requirement as a simple sentence in everyday or business language. An example of this is as follows:

As	User
I want to	manually co-evolve the models State Chart and DTMC
to	to use the DTMC as quality model for my system

Each of the user stories describes a part of a requirement and then they are used as backlog items. This enabled us to be flexible at changing requirements, because just the most important user stories are taken into the next sprint. To integrate the user stories in our GitHub environment (see 3.5.2), we generated an issue for each user story and labeled them. Therefore, each of us knew the current state of the user story and of the development progress as well. The links of a requirement to each user story are given through their GitHub id.

In the following sections some information about the customers are given as well as the classification of requirements. Afterwards all identified requirements are presented with their links to GitHub user stories, their acceptance criteria and the stakeholders.

4.1 Customer

Author: Tim Sanwald

The customer of this project was the Reliable Software Systems Group (RSS) from the University of Stuttgart. The following persons were supervisors of this project:

Prof. Dr. Lars Grunske Examiner of the project.

4. Requirements

Sinem Getir Supervisor and customer for reliability analysis, safety analysis, evolution and co-evolution of models. Provided theoretical incremental transformation and some implemented co-evolutions.

André van Hoorn Supervisor and customer for performance analysis. He also provided information about performance solvers.

For the reason that this was a university project and we had no real stakeholders we associated following common stakeholders to the requirements found with the supervisors:

Maintenance engineer a developer which wants to improve or extend the CoWolf framework.

User a user which wants to create, evolve, analyze and co-evolve models.

4.2 Infrastructure

Author: Tim Sanwald

4.2.1 Co-Evolution Project Nature (#2, #3, #4)

Description The user should be able to create a new project with the specific CoWolf Eclipse nature. The nature should enable the project to be displayed in the project explorer, further it should unlock the popup menu in the project explorer to evolve, co-evolve and analysis models. A folder for models should be generated automatically. For the user it should be possible to add and remove the CoWolf nature in a common way, defined by the Eclipse style guide.

Acceptance Criteria The user is able to create a new project, which contains a models folder and the natures to use the CoWolf menu entries for analysis, evolution and co-evolution. It should be possible to remove and add the nature to a project through the popup menu at the project.

Stakeholders User

4.2.2 Custom Perspective (#5)

Description The user should have the possibility to choose a perspective, which is configured to show all commonly used views related to the CoWolf project. The perspective should include views for navigation on the left, views for console and problems on the bottom, views for outlines on the right.

Acceptance Criteria The perspective can be selected through the Eclipse Perspective Bar. On activating the CoWolf perspective, commonly used views should be opened and the appearance should look like described above.

Stakeholders User

4.2.3 Product (#300)

Description In Eclipse it's complicated to update and install all dependencies of a tool through an update site. In fact of this, it should be possible to download an individual package of the tool, like "Eclipse for Java EE Developers", as zip which includes all dependencies of CoWolf. This is an usability improvement for inexperienced users of Eclipse, so they can use CoWolf without configuring the system.

Acceptance Criteria It should include a zip file, which consists of Eclipse and CoWolf with all dependencies. At start the CoWolf perspective is set as the default perspective.

Stakeholders User

4.2.4 Co-Evolution Controller (#15)

Description To provide the possibility of incremental transformations, the co-evolution have to be done in a defined way to prevent coded clones. The following steps are part of this:

1. Let the user choose the target models
2. Calculate the differences in the source model
3. Call the specific co-evolution on all target models
4. Calculate the differences in the target models and present them to the user

Acceptance Criteria The user can run the Co-Evolution on a model and just have to choose the target models. The user also can follow the progress and gets a result view presented, which displays the changes of the target models. For a Maintenance engineer the effort of developing a new incremental transformation should be as minimal as possible.

Stakeholders User, Maintenance engineer

4.3 Models

Author: Tim Sanwald

4.3.1 Model Definitions (#20, #21, #22, #23, #24, #25, #82)

Description To use CoWolf as a modeling tool the following types of models must be implemented as Ecore Models:

- | | |
|---------------------|--------------|
| ▷ State Chart | ▷ CTMC |
| ▷ Component Diagram | ▷ Fault Tree |
| ▷ Sequence Diagram | ▷ LQN |
| ▷ DTMC | |

4. Requirements

Acceptance Criteria The user can create the models above and store them in the CoWolf project. The Model Instances should contain the commonly used object and references, just some unimportant features can be ignored.

Stakeholders User

4.3.2 Model Validation (#81)

Description The user has to be sure, that his generated model is valid and is not missing any values. To guarantee the validness of a model Object Constraint Language (OCL)s should be provided. To perform a validation, the context menu have to contain an item which performs the validation of the selected model. Once the errors in the model are identified, they need to be marked and should contain an informative description.

Acceptance Criteria The user can choose "validation" at the navigators context menu, which runs a syntactic validation of the model and performs OCL checks to ensure that the co-evolution of this model can be run. The results are marked in the editor and in the "Problems View" of Eclipse.

Stakeholders User

4.3.3 Export Reliability Models (#4)

Description The analysis of DTMC and CTMC is provided within the requirement 4.4.1, but sometimes the user wants to calculate more specific details in PRISM model checker [KNP11] directly. To support external analysis of DTMCs and CTMCs the models should be exportable through the Eclipse export menu. By means of this functionality, the model has to be transformed into a readable input model for PRISM model checker.

Acceptance Criteria The user can find an entry in the export menu of Eclipse, which signalizes the export of a reliability model to PRISM format. By clicking on this entry a new wizard should be opened, so that the user can define the model, if the corresponding Probabilistic Computation Tree Logic (PCTL) rules should be exported too, and the destination of the export. On finish the model and if checked the PCTLs are transformed into a PRISM readable format.

Stakeholders User

4.4 Analysis

Author: Tim Sanwald

4.4.1 Reliability Verification (#17, #159)

Description For calculating the reliability of a system, the user needs to analyze a DTMC or CTMC with PRISM model checker. In order to do such analysis, the UI must provide

a comfortable way to define the properties for the calculation and present the results afterwards.

Acceptance Criteria CoWolf supports the verification and the simulation method of PRISM model checker. For DTMC analysis it's enough to calculate the reachability of states and labels. For the CTMC analysis it is necessary to provide more configuration parameters. It should support some configured examples and analyze PCTLs created by the user. To do this the user should configure the analysis with a wizard and get the results afterwards.

Stakeholders User

4.4.2 Performance Analysis (#18, #77)

Description CoWolf must support the analysis of the performance through LQN performance analysis. The user should run it in the same way as in 4.4.1. The user needs to define the path of the external tool which is used for the analysis.

Acceptance Criteria The user is able to configure the path to LQN Solver [Fra+13] and select the analysis in the context menu. The LQN model is transformed into the input format of LQN Solver and the analysis is called. After finishing of the analysis the user get the results of the performance analysis. The analysis includes the calculation of Throughput, CPU-Usage and Response times.

Stakeholders User

4.4.3 Safety Analysis (#19)

Description To do safety analysis directly from Eclipse CoWolf should support the analysis of safety models. The program call should be as in 4.4.1 to allow the user a high usability, in the background the XFTA tool should be used for analysis.

Acceptance Criteria The user is able to configure the path to the XFTA tool which should be used for analysis. The analysis should contain calculation of the probability that the top event occurs and of all minimum cutsets with it's respective probability.

Stakeholders User

4.5 Editors

Author: Tim Sanwald

4.5.1 Textual Editor (#8)

Description In order to edit the models defined in 4.3.1 the user should be able to create and change the models in a comfortable way. Thus, CoWolf must support a textual

4. Requirements

editor, which supports all functions defined by the Ecore Model to let the user easily create a valid Model Instance.

Acceptance Criteria A textual editor exists for each Ecore Model defined in 4.3.1 and support add, delete or edit operation.

Stakeholders User

4.5.2 Graphical Editor (#9, #10, #38, #183, #184, #185, #186, #187)

Description Some models are too complex or too big for working with a textual editor. In such cases a graphical editor should be implemented for each of the in previously defined models (cf. 4.3.1).

Acceptance Criteria For every supported model a graphical editor is provided which supports the most common used add, delete and edit operations of the model. The representation of the models must be similar to the illustrations in scientific papers.

Stakeholders User

4.5.3 Side-by-Side Editing (#10)

Description In the practical use of a modeling tool, most changes are done in a graphical editor, just because it's more comfortable and easier. Nevertheless sometimes the user wants to do an operation which is not supported by the graphical editor. In fact of this it should be possible to work with both editors on the same Model Instance.

Acceptance Criteria The user can open both editors, graphical and textual, simultaneously and if he edits the model in one of them and saves, the changes are reflected in the other editor.

Stakeholders User

4.6 Evolution

Author: Tim Sanwald

4.6.1 Difference between Models (#16, #128)

Description Changes in the development of a model must be shown with the differences obtained from SiLift to inform the user about changes since the last version. Also, it should be possible to see the complete evolution of a model, which can be achieved by calculating the differences between succeeding versions of the model. Thus, the user can verify the co-evolution manually and get informed about the changes in the models.

Acceptance Criteria The user can choose two or all versions of a model and start the difference calculation which is done in the background. Afterwards the results are presented to the user.

Stakeholders User

4.7 Co-Evolution

Author: Tim Sanwald

4.7.1 Unidirectional (#132, #237)

Description Unidirectional co-evolution is used if the reverse direction makes no sense or is not needed. To proof the concept of unidirectional co-evolution based on the CoWolf framework, incremental transformations for the following pairs shall be explored and implemented:

- ▷ Sequence Diagram to LQN
- ▷ Fault Tree to CTMC based on [BCS07]

Acceptance Criteria The user can create a valid model and run the co-evolution through an entry in the context menu. A wizard appears which allows the user to select the targets for the co-evolution. Only possible targets for the transformation should be selectable. The directions which are possible are characterized in the description. On finish, the co-evolution is called and the specified target models evolves to fit the source model. The user is informed about the changes of the target models.

Stakeholders User

4.7.2 Bidirectional (#13, #131)

Description In contrast to Section 4.7.1 some transformations should be possible for the reverse direction, too. Thus, the following combinations should contain co-evolution in both directions:

- ▷ State Chart and DTMC
- ▷ DTMC and CTMC

Acceptance Criteria Analog to the acceptance criteria in Section 4.7.1, except that it must be fulfilled in both directions.

Stakeholders User

4. Requirements

4.7.3 Integrating Existing Rules (#130)

Description In addition to unidirectional and bidirectional co-evolutions it should be possible to easily integrate existing rules. Thus, the transformation from Component Diagrams to Fault Trees from Ensure project should be integrated into CoWolf to demonstrate the concept.

▷ Component Diagram to Fault Tree

Acceptance Criteria Analog to the acceptance criteria in Section 4.7.1.

Stakeholders User

4.7.4 Inconsistency Detection (#14)

Description CoWolf should prevent inconsistencies in model relations. To do this it's necessary to inform the user about models which may be outdated. This means an associated model has changed.

Acceptance Criteria The user gets informed with Eclipse markers, whenever an associated model of a model changes and a co-evolution is required to keep consistency between the models. The user can then decide whether or not he wants to update the target model.

Stakeholders User

4.8 Maintenance Preparations

Author: Tim Sanwald

4.8.1 Architecture Design (#1)

Description For a well documented framework it's mandatory to have an up-to-date architecture of the system. This reduces the time new developers need to become familiar with CoWolf and illustrates structural problems before they are implemented. Thus, it improves the maintainability and usability of the system.

Acceptance Criteria The system's architecture is represented in a component diagram. Furthermore, the model shows the Eclipse plugin based architecture, the relation between models, the evolution components and co-evolution components.

Stakeholders Maintenance engineer

Foundations and Technologies

To develop CoWolf we used some secondary foundations and technologies to support us. First of all we used the Eclipse Modelling Framework to create and edit the meta models. Further we used Sirius for all graphical editors. To transform the models we needed SiLift and SiDiff for the differences and Henshin for the transformation rules. To be able to analyse the models we used PRISM for CTMC and DTMC, xFTA for Fault Trees and the LQN Solver for LQN models. A detailed description of all used tools follows in this chapter.

5.1 Eclipse-Plugins

We used several Eclipse plug-ins for our development, i.e., EMF, xText and Sirius. They are described in the next section.

5.1.1 Eclipse Modeling Framework

Author: Jonas Scheurich

To support the different models in the eclipse environment an implementation of the meta model, the edit operations and a special editor is required. The Eclipse Modeling Framework delivers a graphical or tree view based editor to create meta models.

Ecore Meta Models

Meta models created with EMF are called Ecore Metamodels. This meta models are stored in the xmi format. Meta models can be created by a treeview editor or a graphical editor (see Figure 5.1).

- ▷ **Class:** A class describes an object in the model instance and contains attributes and references. Classes are able to inherit from other classes.
- ▷ **Attribute:** An attribute contains a value of a certain type. The value type is a self-defined class or a Java type.
- ▷ **Reference:** An reference refers to another class, defined in Ecore Metamodels. Every object instance must be contained into a super object (except the root element). The

5. Foundations and Technologies

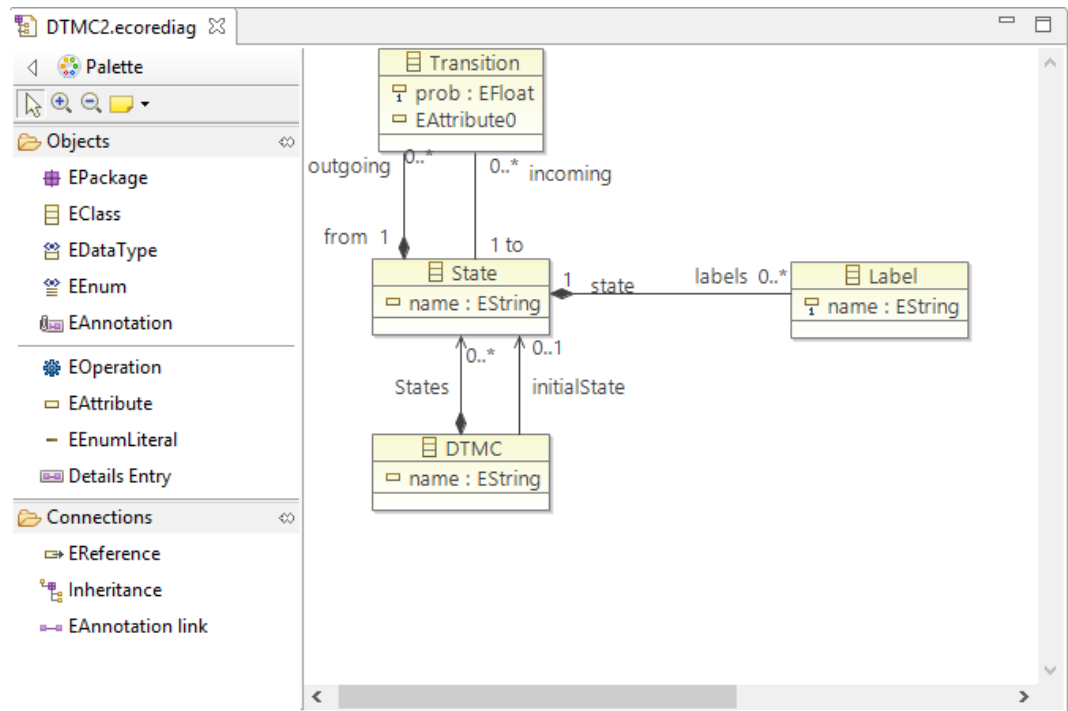


Figure 5.1. EMF editor with a CTMC meta model.

containment property of a reference describes that the target object of the reference is contained in the source object of the reference.

OCL Validation

Ecore Metamodels can be annotated with OCL constraints. The annotations can be created with the EMF editors. But they don't support live validation of the constraints. The eclipse plugin *OCLinEcore* [OCL14] delivers a textual metamodel editor based on xText. This editor supports live validation for the OCL statements. OCL statements can be placed in every class to test constraints of the class or the subclasses. They are executed in the validation process of the treeview or graphical editors. In this validation process the multiplicities of the references and data types are validated, too. The results are shown in the errors view of eclipse and with markers on the model objects in the editor.

Java Code Generation

The main feature of EMF is the code generation from an Ecore Metamodel. An EMF genmodel defines properties such as the file extension, project and package IDs, etc. The

following projects are generated from a meta model:

- ▷ **Metamodel Implementation:** The implementation contains Java interfaces and classes for each Ecore class. The Java classes contains fields and lists in accordance to the Ecore classes. To instantiate Java objects of the generated classes EMF generates a factory.
- ▷ **Edit Operations:** The edit project contains a provider class for Ecore class to perform changes on a model instance.
- ▷ **Model Editor:** The editor project contains the treeview editor with all necessary resources.
- ▷ **Test Stubs:** The test project contains stubs to create own test cases.

5.1.2 Xtext

Author: David Steinhart

Xtext is an open-source framework for language development and defining domain-specific languages. When a DSL is defined, Xtext automatically creates a parser and a customizable Eclipse-based IDE. This also includes a standalone text-field editor which contains live validation for the DSL. When errors in the DSL-instance are found, these are highlighted in the editor. The editor supports auto-completion and syntax highlighting.

Xtext is being developed in the Eclipse Project as part of the Eclipse Modeling Framework Project and is licensed under the Eclipse Public License. [Xte]

5.1.3 Sirius and GMF

Author: Johannes Wolf

Sirius is an Eclipse project which is created by Thales and Obeo [Sir]. Sirius uses GMF for the graphical representation and is a fast way to create graphical editors for domain specific models. It is already used by several modeling tools like the Obeo UML designer. In our case we couldn't use the Obeo UML designer because we used self defined meta models except for the sequence diagram.

The advantages of Sirius are, that you can define a graphical editor directly in your workbench and it will be interpreted during runtime. So you can see a direct feedback while developing the graphical editor without compiling (except for the extension with own Java code). Figure 5.2 shows the workbench for developing a graphical editor with Sirius.

With Sirius it is possible to define node-link diagrams, sequence diagrams, tree views and tables. The definition of the editor is contained in a single .odesign file and no code creation is necessary. It is also possible to use self defined Java services or extend the editor with external components.

Sirius also provides synchronization between the graphical editors and the treeview editor of EMF. This is necessary to provide consistency in the model while editing it in

5. Foundations and Technologies

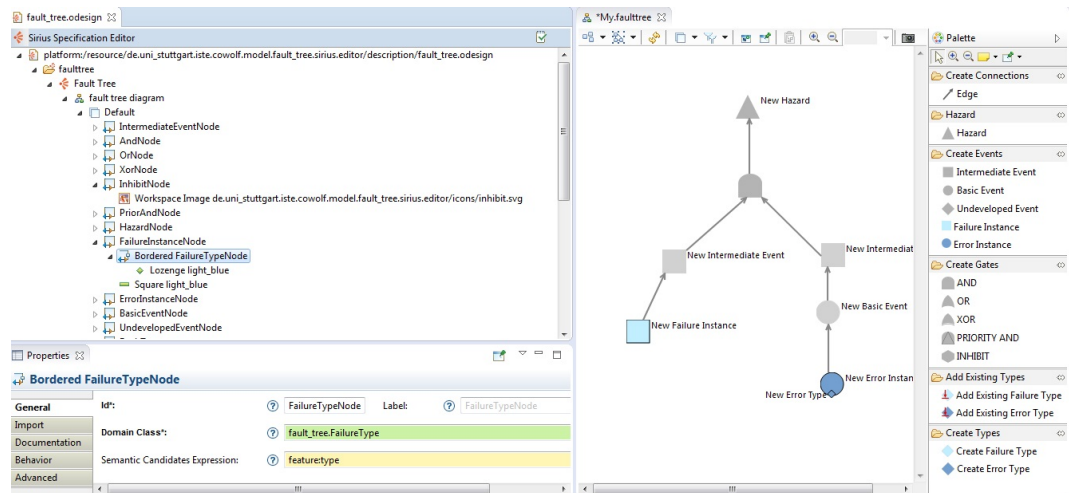


Figure 5.2. The Eclipse workbench with the Sirius editor definition on the left side and the graphical feedback on the right side

different editors. Additionally, Sirius is able to display problems which are caused by OCL constraints directly in the graphical editor [Sir].

5.1.4 Henshin

Author: David Krauss

Henshin is a transformation language based on Ecore with rich graphical tool support. It is part of the Eclipse Modelling Framework. Transformations are represented by Henshin rules and control units, which control the execution of multiple rules. The procedure of transforming a graph is to find a match of a pre-defined structure and replace it with the desired structure. In general, it is either possible to apply transformations on a single model (endogenous) or on at least two models (exogenous). CoWolf makes use of many exogenous transformations when co-evolving models.

Transformation Language

In order to specify a transformation there are positive and negative application conditions (PACs and NACs).

NACs are:

- ▷ «forbid» The element must not be present in the left hand side graph.
- ▷ «require» The required element has to be contained in the left hand side graph.

PACs are:

- ▷ «preserve» An element is contained in both, the left and the right hand side graph.
- ▷ «create» The annotated element is not contained in the left hand side, but created in the right hand side.
- ▷ «delete» The delete tag means that an element is matched in the left hand side graph and is deleted, so it is not in the right hand side graph anymore.

In a Henshin rule, one of these application conditions has to be assigned to each edge, node and attribute that is represented in a rule to define the left and right hand side of the transformation. Typically there are two Henshin files, that represent an arbitrary number of Henshin units. One for the transformation definition and one for its graphical representation. Instead of editing a transformation graphically, the Henshin file can be edited in a tree view, too.

One of the main features of CoWolf is the co-evolution among different models. This is done by finding out the source model's differences and applying the appropriate changes to a corresponding target model by transformations. To achieve this, CoWolf needs to maintain a set of traces between models. A trace is an element that connects two arbitrary EMF-Objects. When transforming changes from a source to a target model, CoWolf knows where to apply the changes at the target model.

Problems with Henshin 1.0

There are several problems using Henshin 1.0:

- ▷ The graphical editor of Henshin causes intensive computing when changing and visualizing large diagrams.
- ▷ Copying elements sometimes destroys the previous layout. This is why large Henshin files have to be handled carefully.
- ▷ Make sure that the model you are transforming is an instance of the same meta model, which you defined the transformation for. If this is not the case, there may occur errors during the transformation.
- ▷ Henshin units are suitable to control the sequence of multiple transformations to a certain extent. If there are complex transformations with multiple rules, it might be advisable to control the flow of Henshin rules by java code. Henshin control units may become difficult to maintain, because changing a big set of units is an uncomfortable activity.
- ▷ Accessing the order of multi references is possible by using the *[i]* syntax. You can access the last element of a multi reference with the index -1, however it is not possible to move elements to this index. This is a problem if you want to iterate through a multi reference list, from the first to the last element.

5. Foundations and Technologies

5.1.5 SiDiff

Author: Michael Zimmermann

CoWolf needs to find correspondences and differences between two model versions. SiDiff [Sof14a] is a tool that offers these functionalities. It was implemented by the Software Engineering Group of the University of Siegen. SiDiff is realized upon OSGi and is available as Eclipse plug-in. An advantage of SiDiff is the possibility to adapt it to your own needs. For example, there are different matchers available to compute the correspondences of two model versions [Sof14c]:

- ▷ *ID-based matcher* which only matches if elements of both models have equal UUIDs.
- ▷ *Signature-based matcher* that uses the signatures of elements. These signatures are calculated based on the attribute values and the relations of the elements.
- ▷ *Similarity-based matcher* that calculates similarities between model elements (e. g., based on their attributes). Details of what should be used to compute the similarity and how it should be weighted can be configured.

The result of the matchers are correspondences between elements of a model A and elements of a model B. After computing the correspondences a difference engine is executed searching the elements of a model that are not part of a correspondence. Thus, created, removed or changed objects, references or attribute values can be detected. The thus determined difference can be seen as a low-level or technical difference and is called symmetric difference in the SiDiff context. Other tools, for example SiLift, can make use of this calculated difference as input for their own work. SiLift will be presented in the following section.

5.1.6 SiLift

Author: Michael Zimmermann

A technical or low-level difference of two models can be hard to understand, especially if more than one element was created, removed or changed. Hence, in order to help developers to comprehend changes in a model the low-level difference needs to be treated to be easily understandable.

A tool that can semantically lift the low-level changes onto a higher abstraction level is SiLift [Keh+14a]. Semantic lifting means, that the low-level differences of the models are lifted into representations of user-level edit operations. Instead of a lot of technical differences between two models only, SiLift can show which edit operations were done by the user. Thus, SiLift makes it easier to understand model evolution. As SiDiff, SiLift is also developed by the Software Engineering Group of the University of Siegen and available as an Eclipse plug-in. Edit operations are defined and detected using Henshin rules. Figure 5.3 shows an example edit rule for the Continuous-Time Markov Chain (CTMC): The Henshin rules first tries to match a CTMC element and creates (if found)

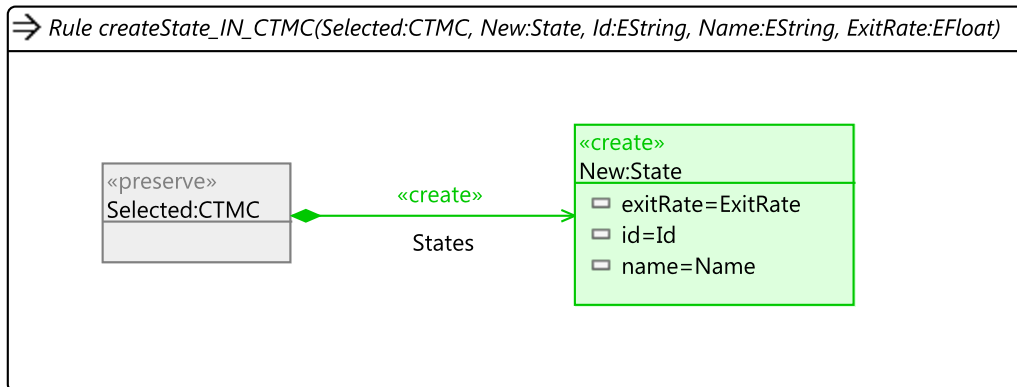


Figure 5.3. Edit operation defined as Henshin rule. It describes the creation of a *State* in a *CTMC*.

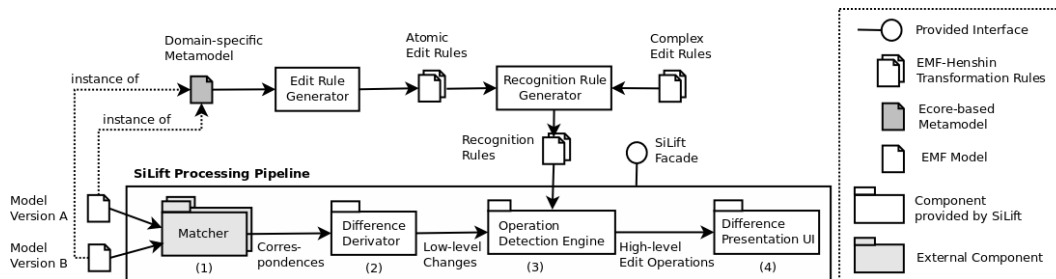


Figure 5.4. Processing pipeline of SiLift [Keh+14b].

a new *State* with the specified attributes in it. This small example edit rule contains all implicit low-level changes like creating the attributes *exitRate*, *Id* and *name* as well as creating the new reference *States*. SiLift distinguishes between atomic and complex edit rules. Atomic edit rules can't be reduced into smaller rules. Complex rules, however, consist of two or more atomic rules and are able to lift the differences on an even higher abstraction level than the atomic rules. In UML, a complex rule would be for example, a refactoring operation *pullUpAttribute*, that deletes an attribute in all subclasses and adds it in the superclass. Atomic edit rules for SiLift can either be created manually or easily generated using the SiDiff Edit Rule Generator (SERGe) [Rin14]. Complex edit rules need to be created manually as it requires a very good model knowledge.

As the edit rules would change the models instead of detecting the defined edit operations, they need to be transformed into so called recognition rules. Recognition rules are Henshin rules themselves and are automatically generated by SiLift based on the edit rules. They are needed to detect if an associated edit rule was executed.

Figure 5.4 shows the four main steps of the SiLift difference processing pipeline

5. Foundations and Technologies

[Keh+14b]:

1. Matcher

The first step in the pipeline is the model matcher. The model matcher searches for objects in the two model instances that are corresponding to each other. There are different matching engines existing (e. g., the SiDiff engine), but you can implement your own one or adapt an existing engine, too.

2. Difference Derivator

The second step is the Difference Derivator. Here the low-level difference, or so called technical difference is built. The low-level difference consists of objects, references or attributes that are deleted or created, as well as changed values of object attributes.

3. Operation Detection Engine

The next step is the Operation Detection Engine. Here the Semantic Lifting is performed. Therefore, the Operation Detection Engine detects groups of low-level changes, representing a high-level edit operation. The generated recognition rules are used as input here.

4. Difference Presentation UI

Finally there is the Difference Presentation UI, that helps the users to inspect the lifted model differences and thus to comprehend the evolution of the model.

5.2 Model Analyzer

This section describes the external tools we used to analyse the quality of service models. We used PRISM, xFTA and a LQN solver. The sections describe what the solvers do and how we access them.

5.2.1 PRISM

Author: David Steinhart

PRISM is a probabilistic model checker. It is used to perform reliability and performance evaluation. For DTMCs the probability to end up in a failure state can be calculated. Using a CTMC representing the time-based behavior of the software system, the performance of different use cases can be evaluated.

As PRISM is not available as an Eclipse plugin, the user needs to manually install it before he can perform verifications. PRISM offers a graphical user interface and a command line interface; the later one is used to access PRISM programmatically. The DTMCs/CTMCs are transformed to a PRISM-readable format using Xtend. As PRISM does not provide an API, the results are saved to a file and need to be parsed afterwards. [Pri]

```

<?xml version="1.0"?>
<!DOCTYPE open-psa>
<open-psa>
  <define-fault-tree name="A Hazard">
    <define-gate name="A Hazard">
      <and>
        <basic-event name="BE1" />
        <gate name="SUB1" />
      </and>
    </define-gate>
    <define-gate name="SUB1">
      <or>
        <basic-event name="BE2"/>
        <gate name="SUB2"/>
      </or>
    </define-gate>
    <define-gate name="SUB2">
      <or>
        <basic-event name="BE3"/>
        <basic-event name="BE4"/>
      </or>
    </define-gate>
    <define-basic-event name="BE1">
      <float value="0.1" />
    </define-basic-event>
    <define-basic-event name="BE2">
      <float value="0.1" />
    </define-basic-event>
    <define-basic-event name="BE3">
      <float value="0.003" />
    </define-basic-event>
    <define-basic-event name="BE4">
      <float value="0.04" />
    </define-basic-event>
  </define-fault-tree>
</open-psa>

```

```

<?xml version="1.0"?>
<!DOCTYPE xfta>
<xfta>
  <load>
    <model input="fault_tree.xml" />
  </load>
  <build>
    <minimal-cutsets top-event="A Hazard" handle="MCS" />
  </build>
  <set>
    <option name="print-minimal-cutset-rank" value="on" />
    <option name="print-minimal-cutset-order" value="off" />
    <option name="print-minimal-cutset-probability" value="on" />
    <option name="print-minimal-cutset-contribution" value="on" />
  </set>
  <print>
    <minimal-cutsets top-event="A Hazard" handle="MCS"
      output="xFTA_out.txt" mode="write" />
  </print>
</xfta>

```

Figure 5.5. XFTA model and script files

5.2.2 xFTA

Author: Manuel Borja

Developed by the Computer Science Laboratory of the Ecole Polytechnique in France, xFTA [Rau12] is a program which allows to perform a variety of probabilistic calculations and simulations on a Fault Tree model written in the Open-PSA format [ER08]. XFTA takes two files as input: one containing the Fault Tree model and another with configuration parameters and execution setup. Both of them are XML files. In Figure 5.5 a simple example of both Fault Tree model and script files are presented.

An xFTA execution session consists on the following steps [Rau12]:

1. The model is loaded.
2. The model is normalized. That means, the common failure causes are expanded, the constants are propagated, etc.
3. The minimal cutsets of the top event are calculated.
4. Further probabilistic calculations simulations are performed always based on the minimal cutsets.

XFTA allows to performe the following probabilistic calculations:

- ▷ Top-event probability

5. Foundations and Technologies

- ▷ Importance factors of basic events
 - ▷ Marginal Importance Factor
 - ▷ Critical Importance Factor
 - ▷ Diagnosis Importance Factor
 - ▷ Risk Achievement Worth
 - ▷ Risk Reduction Worth
- ▷ Sensivity analysis
- ▷ Safety integrity levels for low demand and high demand system

5.2.3 LQN Solver

Author: Manuel Borja

The LQN Solver from the University of Carleton is a set of tools intended to solve LQN models and perform simulations on it. Furthermore, offers a couple of utility tools which allow to transform the models from one format to another. The following programs are included in the LQN Solver:

- ▷ lqn2x: transforms a model in LQN format to another format and vice-versa. The supported formats are xml, ps an emf
- ▷ lqns: solve a lqn model
- ▷ lqnsim: runs a simulation on the LQN model

In order to perform LQN analysis, CoWolf uses just the analytic LQN solver program (lqns). It receives an LQN model file written in LQN format and writes the results into an output file. In the analysis the LQN Solver calculates:

- ▷ Throughput bounds
- ▷ Mean delay for rendezvous and send-no-reply requests
- ▷ Mean delay for joins
- ▷ Entry service times and variances
- ▷ Distributions for the service time
- ▷ Task throughput and utilization
- ▷ Processor utilization and query delays

The LQN Solver uses the Mean Value Analysis (MVA) algorithm to perform the analysis on the model. Analysis parameters like number of iterations, minimum convergence value, pragmas, etc. can be defined both in the model and as program's arguments. LQN Solver performs a series of measurements iteratively on the model giving as a result a series of calculations associated to the elements of the model [Fra99].

5.3 Logback and SLF4J

Author: Rene Trefft

For logging messages and exceptions in CoWolf the logging framework Logback is used which is intended as a successor of log4j [GPH14]. It provides a better performance and less memory consumption than log4j [GPH14]. Logback is highly customizable and supports many appenders which are output destinations, e. g., the console, files and sockets [GPH14]. The following logging levels [GPH14] are provided by Logback:

- ▷ TRACE: For code tracing of a developer.
- ▷ DEBUG: For diagnosis purposes of a developer or an administrator.
- ▷ INFO: A normal significant message, e. g., the start of a system or a procedure.
- ▷ WARN: A non-regular progress, e. g., the same operation will be executed again.
- ▷ ERROR: Program or operation can't continue; needs user interaction.

Logback natively implements the Simple Logging Facade for Java (SLF4J) which is an abstraction for various logging frameworks. Logback must be used in conjunction with SLF4J. [Sif]

Models

Software systems can quickly become very complex. That's the reason why models get more and more important. They can help to simplify complex processes and connections, and focus on the important points in a certain aspect. Models can describe software processes, requirements, dangers and much more. There are several types of models which focus on just a few aspects each. Therefore to model a whole project, you need several different models.

CoWolf includes different model types which can be divided in two main classes: Architecture Models and Quality of Service Models. This chapter only describes the models themselves, the transformations and implementations are described in Chapter 7 and 9.

6.1 Software Architecture Models

The architecture of a software describes its top-level structure. It divides a software in components and how they interact with each other. In this project we support three architecture model types: component diagrams, state charts and sequence diagrams.

6.1.1 Component Diagram

Author: Jonas Scheurich

Components diagrams are structure diagrams to show the different components of a software system and their connections. The connections are represented by ports and interfaces. Figure 6.1 shows an overview of a component diagram.

6. Models

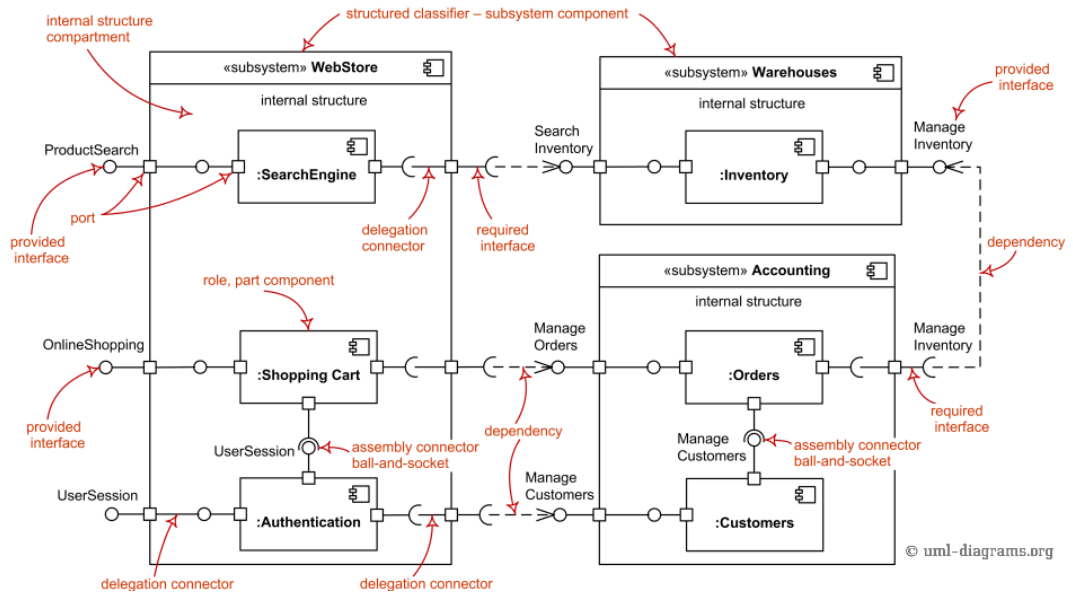


Figure 6.1. Overview of a component diagram. [Sd]

In the following the main elements are described and which of them we implemented in Co Wolf.

Main Elements

Component A component, shown in Figure 6.2 represents a subsystem of a software-system. A component provides ports with interfaces to connect to other components.



Figure 6.2. A component [Sd]

Interface An interface represents connections to other components with well defined functionality.

- ▷ **Provided Interface** A provided interface, (see Figure 6.3) supports functionality of the component. The provided interface can be realized by a required interface. In the following two types of interfaces are described.

6.1. Software Architecture Models

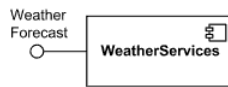


Figure 6.3. A provided Interface [Sd].

- ▷ **Required Interface** A required interface (see Figure 6.4) uses a given interface of a second component.



Figure 6.4. A required Interface [Sd].

Connector A connector connects one or many required interfaces with a provided interface. In the following the two types of connectors are described.

- ▷ **Simple Connector** A simple connector (see Figure 6.5) connects a required interface and a provided interface.



Figure 6.5. A simple connector [Sd].

- ▷ **Multi Connector** A multi connector (see Figure 6.6) connects many required interfaces with one provided interface.

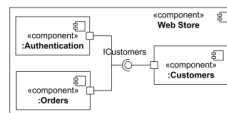


Figure 6.6. A multi connector [Sd].

6.1.2 State Charts

Author: Jonas Scheurich

A state chart is a deterministic machine. A state chart contains states and transitions to visualize the different states of a software system or a sub component of a software system. Figure 6.7 shows an overview of a statechart.

6. Models

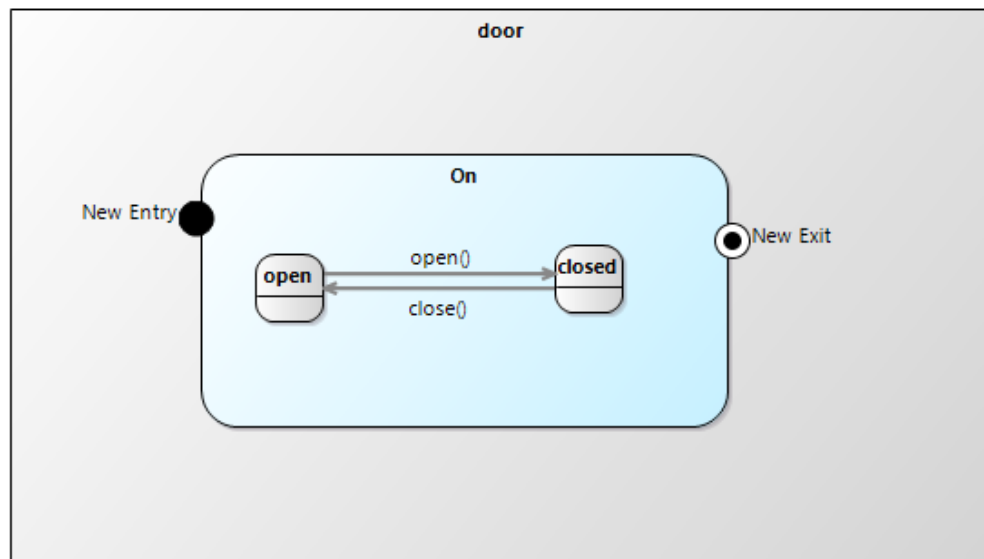


Figure 6.7. Overview of a statechart.

Main Elements

State Machine A statemachine contains transitions and states and represents the software-system. The statemachine also defines the initial state. Figure 6.8 shows an empty state machine.



Figure 6.8. A statemachine.

State A state (see Figure 6.9) represents a state of the software system. The state is active if a transition to the state is activated.



Figure 6.9. A state.

Composite State A composite state (see Figure 6.10) is similar to a state. Additionally the composite state contains some states and transitions. The initial state will be activated if the composite state is activated.

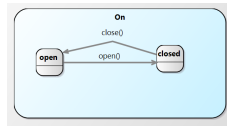


Figure 6.10. A composite state.

Entry An Entry (see Figure 6.11) call to a state machine which is executed on state activation.



Figure 6.11. An entry state.

Exit An Exit (see Figure 6.12) call to a state machine which is executed on state deactivation.



Figure 6.12. An exit state.

Do Action An Action (see Figure 6.13) calls to a state machines if the entry action is done.

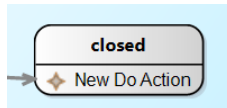


Figure 6.13. A do action.

Transition A transition (see Figure 6.14) deactivates the source state and activates the target state.

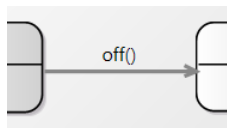


Figure 6.14. A transition

6.1.3 Sequence Diagram

Author: Verena Käfer

Sequence diagrams are interaction diagrams. They focus on the sending of messages between lifelines. Figure 6.15 shows an overview of the possible elements in a sequence

6. Models

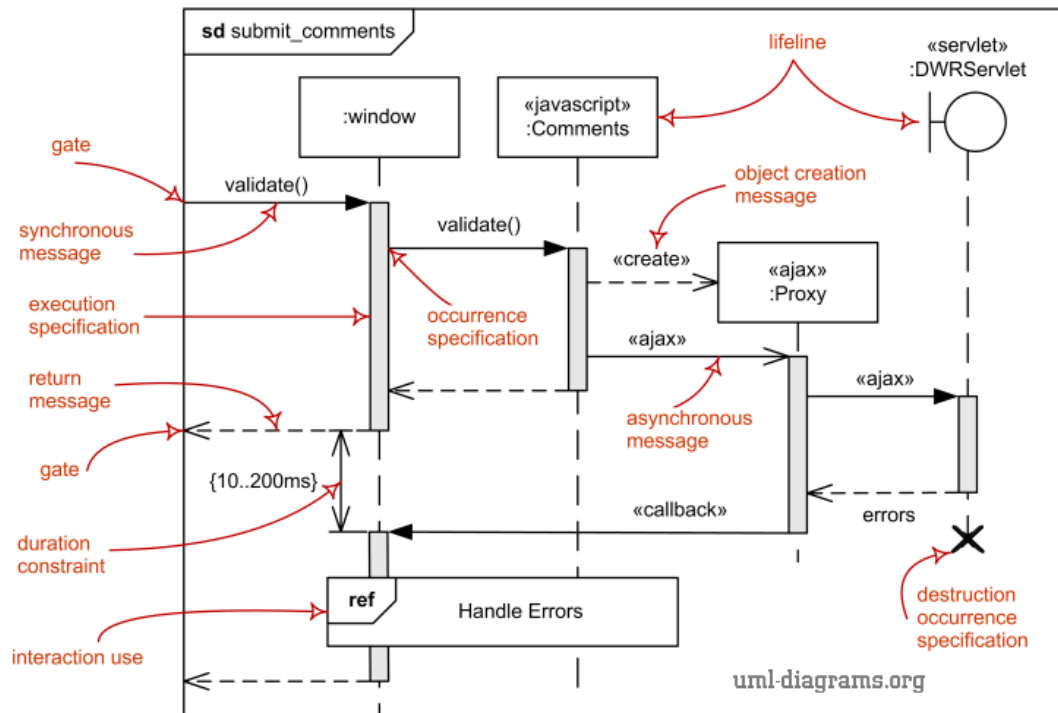


Figure 6.15. Overview of a sequence diagram. [Sd]

diagram.

In the following the main elements are described and explained which of them we implemented in Co Wolf.

Main Elements

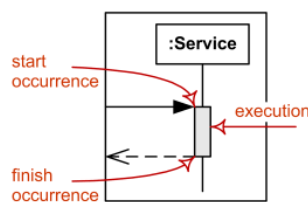


Figure 6.16. A lifeline header [Sd]

Lifeline A lifeline represents an individual participant in the whole interaction. Lifelines (See Figure 6.16) can be divided into class- and actor-lifelines. Actor-lifelines represent an external actor, like a human, who starts single interactions. Class-lifelines represent

class instances and can be called by actor-lifelines or executions of other class-lifelines.

Messages In general, messages define the communication between lifelines. A message starts on a lifeline or an execution and ends in a new execution. There are several message types.

- ▷ **Synchronous Call** This kind of message represents a synchronous call to another lifeline. It always gets an answer (See Figure 6.17). The message starts an execution and at the end of the execution the reply message is sent back. The original sender has to wait for the answer before it is able to send a new message.

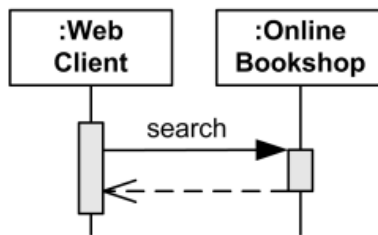


Figure 6.17. A synchronous call with its reply message [Sd]

- ▷ **Asynchronous Call** Asynchronous messages need no reply (See Figure 6.18). They just start a new execution.

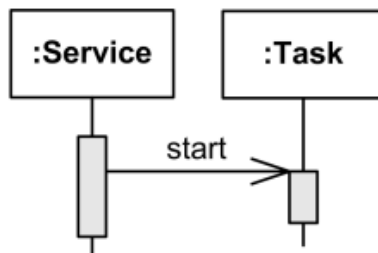


Figure 6.18. An asynchronous call [Sd]

- ▷ **Create Message** This kind of message creates a new lifeline as int can be seen in Figure 6.19.

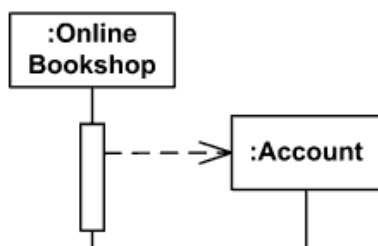


Figure 6.19. A message which creates a new lifeline [Sd]

6. Models

- ▷ **Delete Message** With this type of message the receiving lifeline is deleted as shown in Figure 6.20.

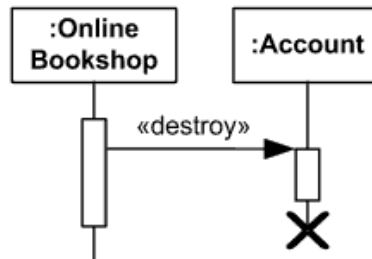


Figure 6.20. A message which deletes the receiver [Sd]

Execution A message always creates an execution on the receiving lifeline (See Figure 6.21). During an execution it is possible to send further messages. An execution may have a reply message at its end.

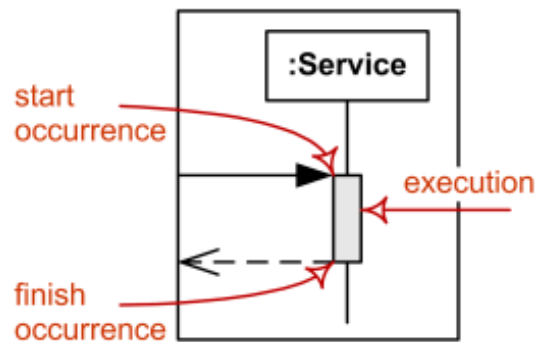


Figure 6.21. An execution on a lifeline with its start and end point [Sd]

Interaction Fragment An interaction fragment is a square in a sequence diagram which specifies a different behaviour than the default behaviour. Following the different types of interactions are described.

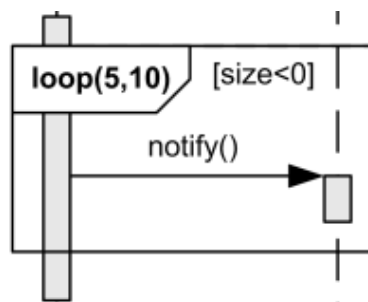


Figure 6.22. A sub-sequence which is executed several times [Sd]

- ▷ **Loop** It is possible to model looping behaviour in an interaction fragment (See Figure 6.22). The fragment contains the looping behaviour and the condition specifies how long the behaviour is repeated.
- ▷ **Alt** Conditional behaviour can be modeled in an alternative fragment as shown in Figure 6.23.

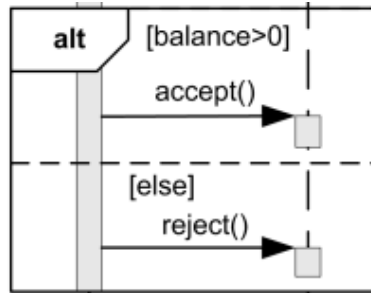


Figure 6.23. Two alternative behaviours depending on the condition [Sd]

Restrictions in CoWolf

In CoWolf sequence diagrams are represented in a graphical editor (Chapter 9.1.2) and transformed into LQN diagrams (Chapter 7.5). To be able to do this, it would have been very hard and time consuming to have all existing elements in our meta model. Therefore we decided to model only the most important elements:

- ▷ Lifelines
- ▷ Synchronous and asynchronous messages
- ▷ Executions

Unfortunately this means that the following elements are not included in CoWolf:

- ▷ Create message
- ▷ Delete message
- ▷ Interaction fragments

6.2 Quality of Service Models

QoS models are intended to represent non-functional requirements of a software system. Thus, it is possible to construct a model with a set of elements which represent real objects

6. Models

or events that affect, for example, the reliability of a system. Once the model is built, it is possible to assess, analyze and predict the quality of the system.

The QoS models supported by CoWolf namely Discrete Time Markov Chain (DTMC), Continuous Time Markov Chain (CTMC), Layered Queuing Network (LQN) and Fault Tree will be presented in this section.

6.2.1 Discrete Time Markov Chain (DTMC)

Author: Johannes Wolf

The discrete time Markov chain model is a model to describe states of a system and the transitions between them. This model is often used for statistical processes with a collection of random variables. The transitions between the states describe the probability that the following state will be reached. The transition depends only on the current state and not on the sequence of the states before. This fact is also called the markov property. The sum of the outgoing transition probabilities of a state has to be always 1. Therefore in each discrete time step a transition has to be executed. It's also possible to have self-referencing transitions.[Gal] In Figure 6.24 you can see the meta model of the DTMC.

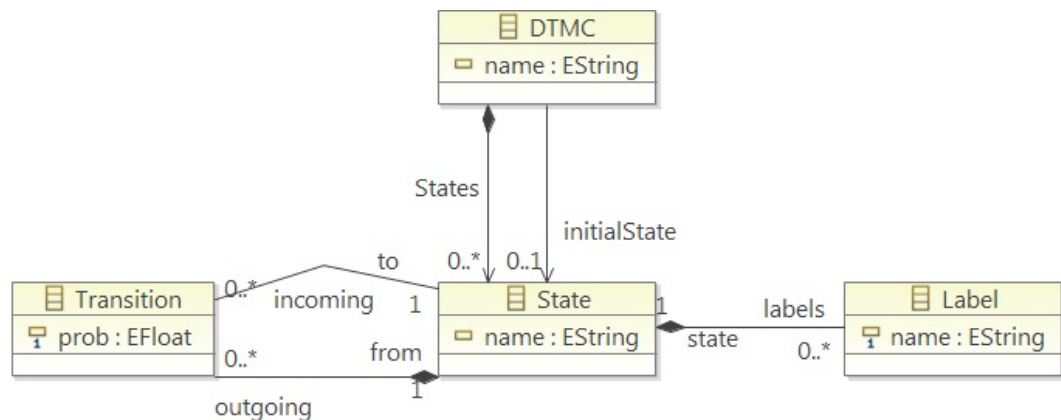


Figure 6.24. The DTMC meta model

Each State can contain one or many labels. This is necessary for the analysis of the model. For example you can define an end state by using a label called *End* and calculate the probability and the amount of time steps to reach this state. To determine the start state, the Ecore class *DTMC* contains the attribute *initialState* which refers to the initial state.

Figure 6.25 shows a simple DTMC model instance which consists of six states. *State E* is labeled as an end state.

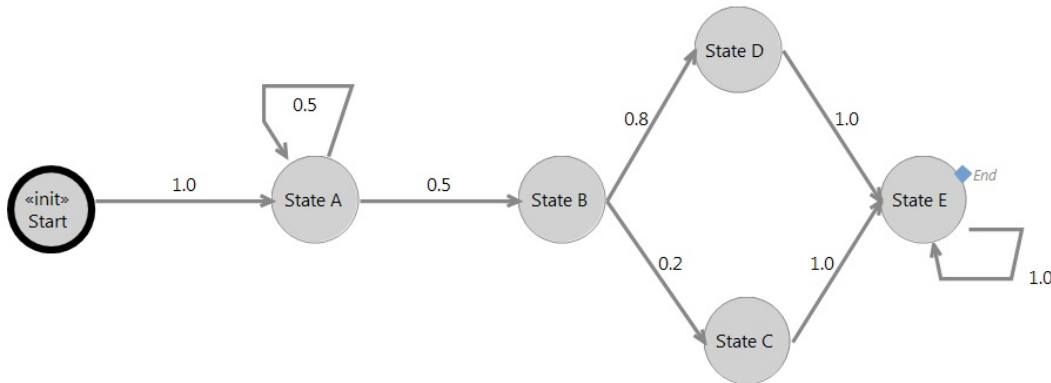


Figure 6.25. Graphical representation of a simple DTMC model in CoWolf.

The same model can also be described in a mathematical form as the following matrix of transition probabilities:

$$P = \begin{pmatrix} P_{Start,Start} & P_{Start,StateA} & \cdots & P_{Start,StateE} \\ P_{StateA,Start} & P_{StateA,StateA} & \cdots & P_{StateA,StateE} \\ \vdots & \vdots & \ddots & \vdots \\ P_{StateE,Start} & P_{StateE,StateA} & \cdots & P_{StateE,StateE} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

6.2.2 Continuous Time Markov Chain (CTMC)

Author: Johannes Wolf

The continuous time Markov chain model is similar to the DTMC model. The difference is, that the time is continuous instead of discrete. As you can see in Figure 6.26 the Ecore class *State* contains additionally the attribute *exitRate* which describes how much time is spent in a state before a transition. The duration for a transition is represented by the attribute *rate* in the Ecore class *Transition* [Sha]. These rates are described by the following transition rate matrix:

$$P = \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots \\ q_{1,0} & q_{1,1} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

In this matrix $q_{i,j}$ describes the probability per time unit that a transition from state i to state j will be executed:

$$q_{i,j} = \lim_{\Delta t \rightarrow 0} \frac{P\{X_{t+\Delta t} | X_t = i\}}{\Delta t}, i \neq j$$

6. Models

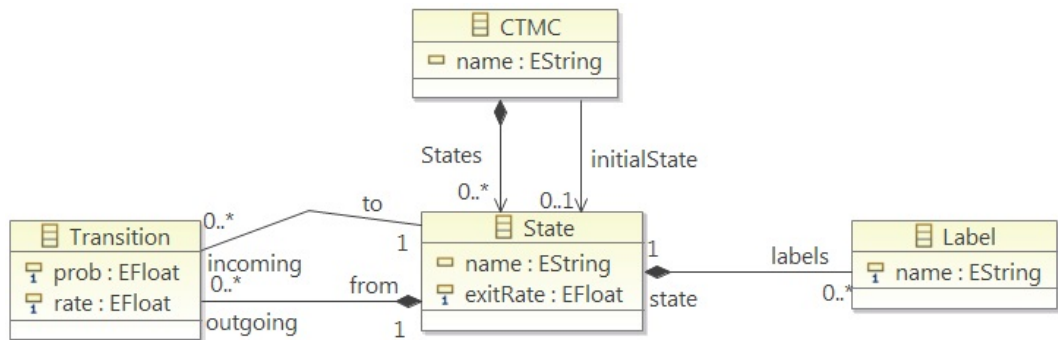


Figure 6.26. The CTMC meta model.

Figure 6.27 shows an example model instance of the CTMC. The exitRates displayed in the brackets inside the states.

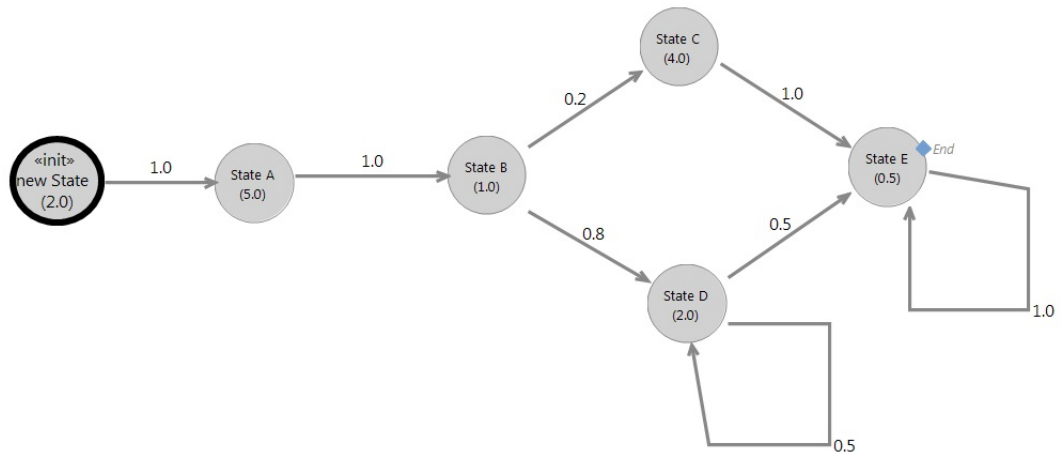


Figure 6.27. Graphical representation of a simple CTMC model in CoWolf

6.2.3 Layered Queuing Network (LQN)

Author: Manuel Borja

Layered Queued Network [Fra+13] belongs to the family of extended queuing networks (EQN). In addition to the standard queuing network models, LQN offers two main capabilities:

- ▷ Simultaneous use of resources
- ▷ Element modeling in a layered structure

LQN is getting good acceptance in many areas especially in software engineering. Techniques like Software Performance Engineering (SPE), make intensive use of queuing models in order to identify problems within a system and assess its performance [DN93].

Elements

The Figure 6.28 shows a simple LQN model instance. It represents a book store system consisting of two clients (administrator and customer) and a software system (the bookstore) which contains a component named ShoppingCart.

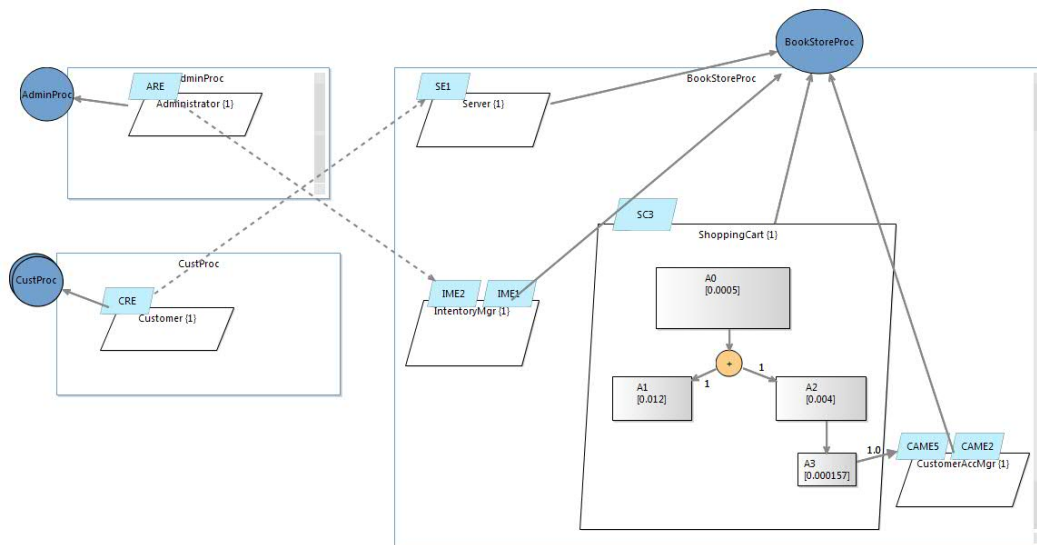


Figure 6.28. LQN example

An LQN model consists of the following elements:

- ▷ **Processors:** they are resources responsible of doing tasks which consume time. As they are pure servers, they only receive requests from other servers or clients. An example for this kind of elements are hardware components like actual CPUs. The Figure 6.29 shows the graphic representation of this component.

The processors are described by the following properties:

- ▷ Schedule discipline
- ▷ FIFO: first in, first out
- ▷ PPR: priority, preemptive resume

6. Models

- ▷ HOL: head-of-line priority
- ▷ PS: processor sharing
- ▷ RAND: random scheduling
- ▷ CFS: complete fair scheduling

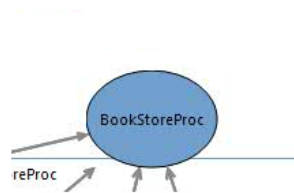


Figure 6.29. Graphical representation of a processor

- ▷ **Tasks:** Offer the possibility of modeling resources. They can be actual tasks in a software system, clients, buffers and hardware devices. In order to specify more details about tasks, different kind of services are modeled with entries. Internal concurrency is modeled with activities graphs. As seen in the Figure 6.30 a task is represented with a parallelogram.

The requests can be attended with one of the following policies:

- ▷ FIFO: first-in, first-out
- ▷ PPR: priority, preemptive resume
- ▷ HOL: head-of-line priority
- ▷ Priority: can have values from zero to positive infinity. Zero is the highest value

Tasks can be classified in three types, namely:

- ▷ Reference class
- ▷ Semaphore task
- ▷ Synchronous task

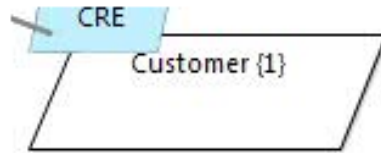


Figure 6.30. Graphical representation of a task

- ▷ **Entries:** They model the actual service request processing. A task can contain many entries and they can have different behavior. i.e. type, different priority and open arrival rate. They can accept either synchronous or asynchronous requests. In the case of synchronous requests, entries must generate a response as well. The Figure 6.31 shows the graphical representation of an entry.

The parameters of an entry can be defined in two ways: phases or activities. Whereas by the phases specification the entry's activities are executed in a sequential way, the activities description allows to model more complex behaviors between the activities including elements like join, loops, fork etc.

Independently of the specification method, the behavior of the server with its clients is made by phase. The first phase contains a complete request-response process. The following phases can be added and take place after the server response.



Figure 6.31. Graphical representation of an entry

- ▷ **Activities:** They are the lowest-level specification of the LQN model. They consume time on the processor and relate with each other through a directed acyclic graph. They make requests to other tasks as well and their behavior can be parametrized with the attribute call order, which can be stochastic or deterministic. Whereas in the deterministic call order the activity makes an exact number of requests, in the stochastic call order the number of requests of the activity is a random number. The Figure 6.32 shows an example of an activities graph.

6. Models

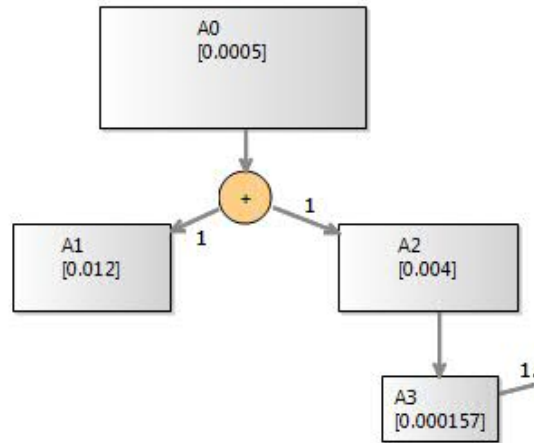


Figure 6.32. Graphical representation of an activities graph

6.2.4 Fault Tree

Author: Manuel Borja

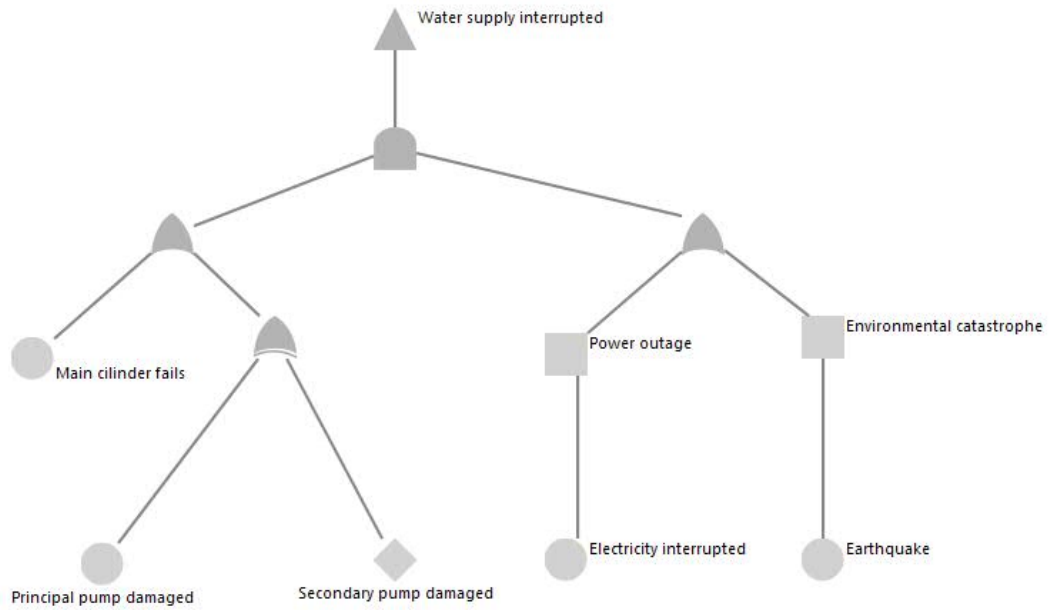


Figure 6.33. Example of a Fault Tree

System safety has been a matter of concern in different fields like aircraft and aerospace industries, which for the first time used this approach in order to find heuristics to determine the causes of accidents before they could occur.

Fault Tree Analysis is a widely used technique to assess safety and reliability. It determines in a deductive way the causes of an undesirable event. Furthermore their relations and quantitative properties are established. The result of the Fault Tree Analysis is a Fault Tree model instance. It shows in a graphical fashion how basic failures combine themselves to produce a hazard. The Figure 6.33 shows a trivial Fault Tree example.

6.2.5 Elements of a Fault Tree

The elements of a Fault Tree are listed as follows:

- ▷ **Hazard:** Also called the top event, it describes the undesirable event to be analyzed. The Figure 6.34 shows the graphical representation of a hazard.

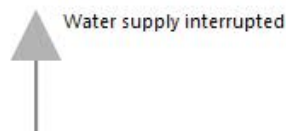


Figure 6.34. Graphical representation of a hazard

- ▷ **Events:** They represent the causes that can trigger the hazard. They can be classified in:
 - ▷ **Primary events:** Represent root causes of the hazard. In the context to be evaluated they have not been further developed. They can be classified again in:
 - ▷ **Basic events:** Represent a basic initiating fault with no further development. The Figure 6.35 shows the graphical representation of a basic event.

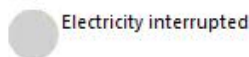


Figure 6.35. Graphical representation of a basic event

- ▷ **Conditioning events:** Represent a condition associated to an event. They are used in conjunction with priority and inhibit gates.
- ▷ **Undeveloped events:** It is an event that is not further developed, because of missing information or because it has insufficient consequences. As can be seen in the Figure 6.36 the graphical representation of an undeveloped event is a rhombus.

6. Models



Figure 6.36. Graphical representation of an undeveloped event

- ▷ **External event:** They represent an event which is always expected to happen.
- ▷ **Intermediate events:** They represent events with particular behaviour. They are used to give further description of the behaviour and development of basic events. The Figure 6.37 shows the graphical representation of an intermediate event.

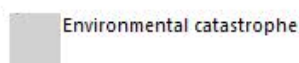


Figure 6.37. Graphical representation of an intermediate event

- ▷ **Gates:** Gates are used to permit (or inhibit) the propagation of the fault logic up the tree. They act like black boxes which receive a set of input events and produce an output event. The gate's logic determines under which conditions the errors from the input events are propagated or not to the output event. Fault Tree model supports the following gates:
 - ▷ **AND:** The error is propagated if and only if all the input events occur. The graphical representation of an AND gate is shown in the Figure 6.38.



Figure 6.38. Graphical representation of an AND gate

- ▷ **OR:** The error is propagated if and only if at least one of the input events occur. The Figure 6.39 shows the graphical representation of an OR gate.



Figure 6.39. Graphical representation of an OR gate

- ▷ **INHIBIT:** It is an especial case of the AND gate. The output is caused under certain condition of the input gate. As can be seen in the Figure 6.40, the INHIBIT gate is represented with a hexagon.

6.2. Quality of Service Models



Figure 6.40. Graphical representation of an INHIBIT gate

- ▷ **XOR:** It is a special case of an OR gate, in which the output is produced if and only if exactly one of the inputs occur. An example of a graphical representation of an XOR gate is shown in the Figure 6.41.



Figure 6.41. Graphical representation of an XOR gate

- ▷ **PRIORITY-AND:** It represents a special case of the AND gate, in which the output is produced only if the input events occur in a specified sequential order. The graphical representation of a PRIORITY-AND gate is shown in the Figure 6.42.



Figure 6.42. Graphical representation of an PRIORITY-AND gate

Transformations

Previously was described, which models are included in CoWolf. Here is described how the transformations between these models work. There are five different transformations of which some are bidirectional and the others unidirectional. The sections describe how the transformations work and what difficulties they contain.

7.1 State Chart to DTMC and Vice Versa

Author: Michael Müller

In opposite to most other transformations (with the exception of the DTMC to CTMC transformation) discussed later in this chapter, the transformation between State Chart and DTMC is implemented bidirectional. As one can see in Table 7.1 the rules which are needed for the transformation between both diagram types are in all cases a simple bidirectional one-to-one mapping, where traces between elements of different diagram types could easily be created.

The transformation is implemented only for the topmost states and transitions of a State Chart model. States and transitions which are part of a composite state are not considered in the transformation. This doesn't match for Action elements (Do, Entry, Exit), which can call another sub-statemachine. For these actions it is needed to find the first parent state which got transformed to DTMC and connect it by a transition to the initial state of the called sub-statemachine.

The transformation DTMC to State Chart has essentially the same transformation mappings applied as described in Table 7.1. Additionally, some restrictions apply here:

State Chart	DTMC
State	State
Composite State	State
Transition	Transition
Action (if it has a StateMachine call)	Transition
Label	Label

Figure 7.1. Transformation mapping State Chart ↔ DTMC

7. Transformations

- ▷ Created transitions in DTMC always create a transition in the State Chart model. There is no possibility to create a transition in the DTMC which is mapped to an action in the State Chart model.

- ▷ Created states in DTMC always map to a state in the main sub-statemachine in the State Chart model, if there is no existing trace to a state elsewhere. It is not possible to transform a created state into another sub-statemachine.

A Problem of this transformation is the time-consuming activity of adding the correct probability to each transition automatically, as this information cannot be extracted from a transition in a State Chart model. Additionally, the created DTMC model is far more confusing compared to the State Chart model because it is not possible to arrange the DTMC in several sub-DTMCs (cf. sub-statemachines in State Chart). Also unlike in a State Chart model, transitions can not be named in a DTMC model.

7.2 CTMC to DTMC and Vice Versa

Author: David Krauss, Verena Käfer

The transformation between CTMC and DTMC is the easiest implemented transformation. Every CTMC state equates to a DTMC state, every CTMC transition equates to a DTMC transition and every CTMC label equates to a DTMC label. The only difference is, that CTMC states have an exit rate (For further details on the models see 6.2.1 and 6.2.2). The exit rate is calculated automatically from all outgoing transitions of a state. The probability of a transition is transferred among both model types equally. Furthermore each transition t , outgoing from a state s , in a CTMC contains a rate. This rate is calculated by:

$$t.rate = t.prob * s.exitRate \quad (7.1)$$

Both models have the same elements, changes can be directly applied from source to target model in both directions (CTMC to DTMC and DTMC to CTMC). The mapping from a difference to the execution of a Henshin unit is defined by a transmap XML file (see 12.3.2). Following is an example for the one to one relation of both models:

When there is a newly created State in a CTMC model, SiLift will detect a change called *CREATE_State_IN_CTMC*. The transformation then looks at the CTMC-DTMC mapping XML file and executes the specified Henshin unit called *CreatedStateInCTMC*. This Henshin unit creates a new state in the DTMC target model, which is correctly connected to its source state in the CTMC by a trace object. Every difference in the source model, that has an impact on the target model is mapped to a Henshin unit in the transmap file. Thereby both models are kept well consistent.

7.3 Fault Tree to CTMC

Author: David Krauss, Michael Müller

The transformation from Fault Tree to CTMC is an unidirectional transformation. Using this transformation it is possible to analyze a Fault Tree regarding the reliability of the described system.

Some general properties apply to this transformation:

- ▷ An hazard creates always two states in the CTMC, the initial state and an absorbing state in the CTMC model.
- ▷ Gates created under another gate need a pseudo state which is replaced by its developed CTMC representation at the end of the transformation process.

The transformation process mainly focuses on the development of the following described patterns. The gate patterns can be divided in two parts. One part comprises the Or and And gate, where the order of occurred events doesn't matter and the other part is the PriorityAnd gate which takes the order of event occurrence into account.

This transformation is available only for one direction because the reverse transformation of these patterns is only possible with a huge amount of effort. Also, one could not rely on the information provided by the user as one single failure in entering transition possibilities or targets could break an entire pattern which represents a gate. Additionally, it is unusual to directly modify an CTMC model (apart from transition rates) rather than only use it for analysis purposes.

7.3.1 Or Gate and And Gate Pattern

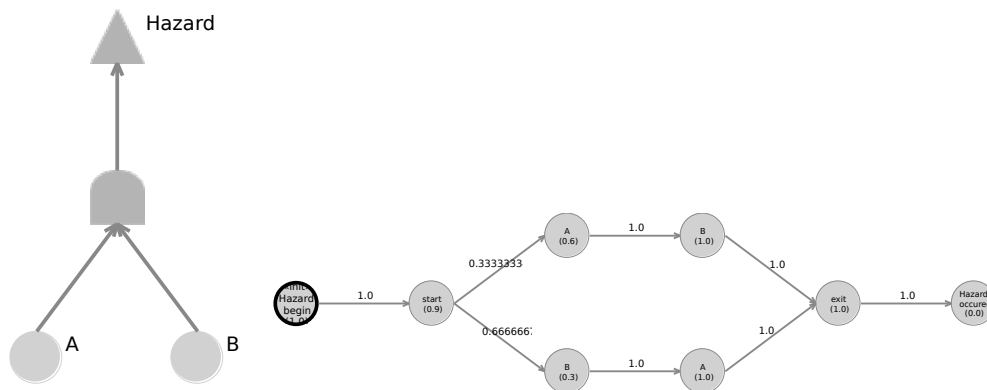


Figure 7.2. And Gate

7. Transformations

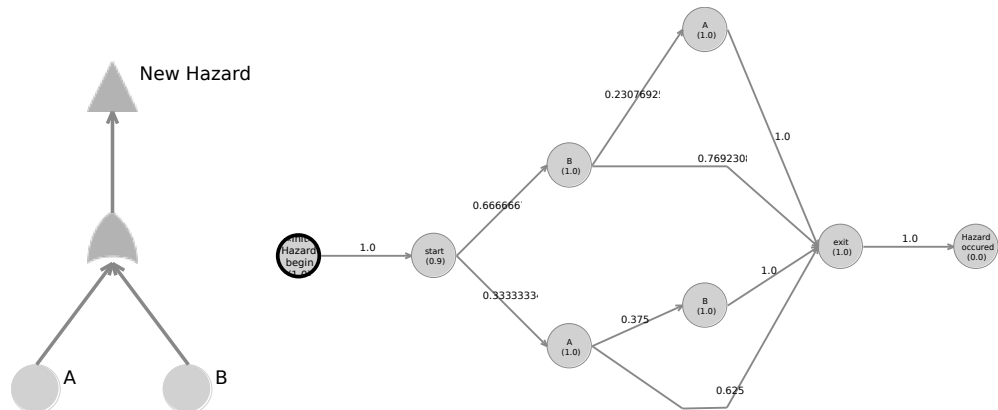


Figure 7.3. Or Gate

Both patterns are very similar in terms of transformation rules. As they both allow every sequence of occurring events to be valid, every permutation of events is created in the CTMC model, as can be seen in Figures 7.2 and 7.3. To ease the transformation process, states which could be visualized as one state are represented as individual states. Merging them would only aid the visualization but the representations are equivalent. Thus the transformation can be implemented with an approach similar to a depth-first search. At the current state it is always known which events already happened. All Events, that didn't occur in the branch up to the current state and additionally are not connected by a transition from the current state, are added to the graph and have a transition is created between the current and the new state.

Compared to the And gate pattern, the Or gate pattern creates an edge from every state, which is not the initial state, to the exit state of the pattern. While the transition probability between states can be directly transformed from the fault tree event associated with the CTMC state, this is not possible for edges from these intermediate states to the exit state. After the transformation process where all single gates are developed independently without knowledge of other existing gates, they are copied and inserted at places where pseudo states were created instead of the gate.

7.3.2 Priority And Gate Pattern

The Priority And gate requires the incoming events (A and B) to occur in a defined order to trigger the top event/hazard. The instance of the gate holds a list of incoming events, which defines the order in which the incoming events have to occur. In Figure 7.4 event A has to occur prior to B, otherwise the hazard will not occur. If B occurs first, there is no path to the hazard state anymore. The implementation has problems when connecting

7.4. Component Diagram to Fault Tree

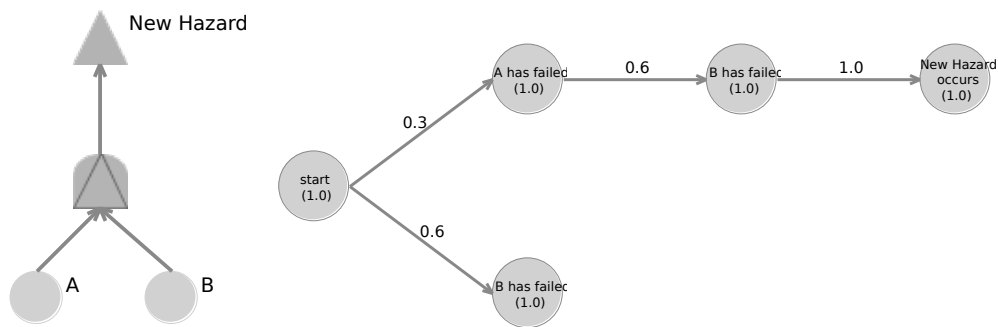


Figure 7.4. Priority And Gate

other gates as input to a Priority And gate, because it is not clear how to find out about the order of input events and input gates. To receive an ordered list of events and gates, the meta model has to be changed first.

7.4 Component Diagram to Fault Tree

Author: David Krauss, Manuel Borja

One of the purposes of transforming a component diagram model into a Fault Tree model is to represent and assess the safety of the system described in the architectural model. This is made by establishing which components can be point of failure, and how they propagate a failure along the remaining components. The implementation of this transformation has been carried out unidirectional only because changes in the component diagram necessarily result in changes in the Fault Tree but not the other way round. If there is any new component, there can always be a new failure or an error event. If any of the components is removed, it obviously cannot fail anymore, so the corresponding events cannot occur and have to be deleted accordingly. Changes in the Fault Tree however, do not necessarily have an impact on the structure of the system. If there is a new event in the fault tree, we do not want to create a new component in the component diagram in general and there is no reasonable mapping that could apply for any change principally.

7.4.1 Implementation

Unlike the other transformations supported by Co Wolf, whose rules and mappings are implemented totally in Henshin, the transformation patterns for the co-evolution from component diagram to fault tree have a big part of its logic written in Java and are supported by a set of granular Henshin rules. In a standard co-evolution the following tasks are performed:

7. Transformations

- ▷ Read the component diagram model.
- ▷ Read the Fault Tree model.
- ▷ Identify new elements in component diagram model.
- ▷ For every new element and based on the transformation patterns, determine which transformations need to be performed and then execute it. As a result there will be new events, gates and relations between them in the Fault Tree model.
- ▷ Based on the new elements in the Fault Tree, perform further transformations to keep the fault tree model consistent.

7.4.2 Transformation Patterns

New Component Instances

New component instances in the component diagram lead to new error or failure instances and new basic or intermediate events in the Fault Tree model.

This transformation pattern describes the way a component instance in the component diagram model has to be reflected in the Fault Tree model. As components can be points of failure in a system, every component corresponds to either a basic or an intermediate event. Furthermore they are associated with either an error type or a failure type respectively. Figure 7.5 shows an example of this transformation among with the *new connections* pattern described in the following section.

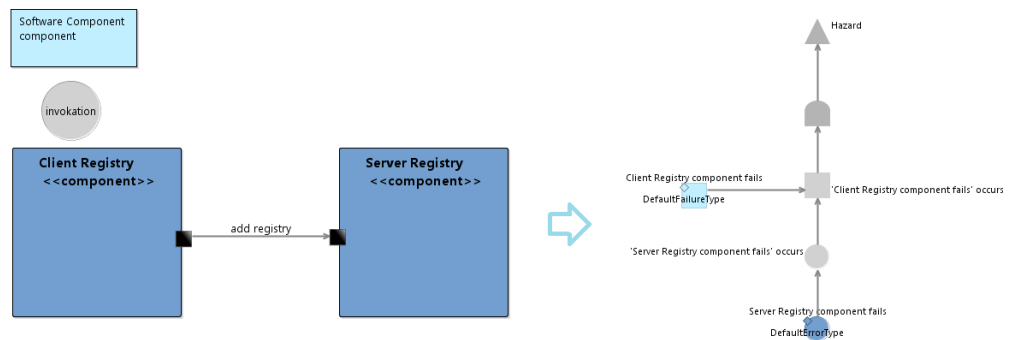


Figure 7.5. New component instances pattern

New Connection between Components

New connection in the component diagram leads to new connection between events in the Fault Tree model.

7.4. Component Diagram to Fault Tree

This transformation pattern treats new connections between two component instances in the component diagram. In principle, a new connection between two component instances (out-port to in-port) intrinsically establishes an unidirectional dependency between them. If the independent component fails, the error is propagated to the dependent component, i.e., the dependent component fails, too. Thus, in the safety model the dependent component must correspond to a basic event and the independent event must correspond to an intermediate event. Figure 7.6 shows an example of this pattern.

Here, the complexity resides in the fact that there might be intermediate or basic events which are already connected to gates or another intermediate events. Nevertheless, the logic of the propagation error is maintained and eventually basic events will be transformed into intermediate events.

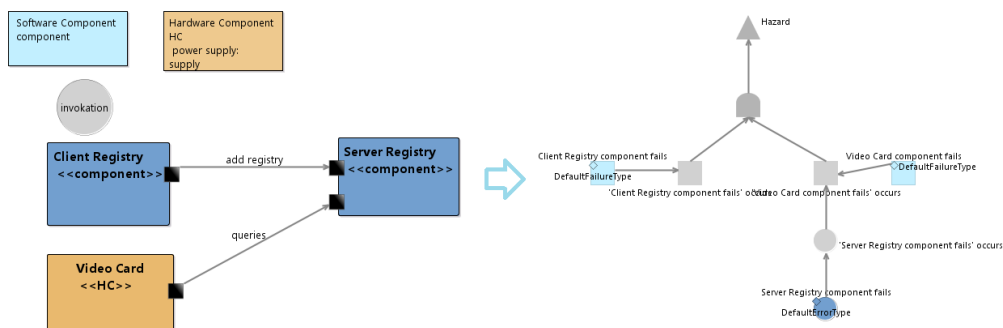


Figure 7.6. New connections pattern

New Connection between Components II

New connection between software component and sensor component leads to top intermediate event.

This pattern is applied when there is a new connection between two components which are of type sensor and software component. These two components need to be contained by the same parent component in order to apply the pattern. It creates two events at the Fault Tree side, corresponding to the sensor and the software component. Both are connected to an OR gate with an output intermediate event on top of it. Whenever there are already existing events at the Fault Tree side, which might be already connected to any other element, the pattern needs to adapt to the current situation. Figure 7.7 shows an example of this pattern.

7. Transformations

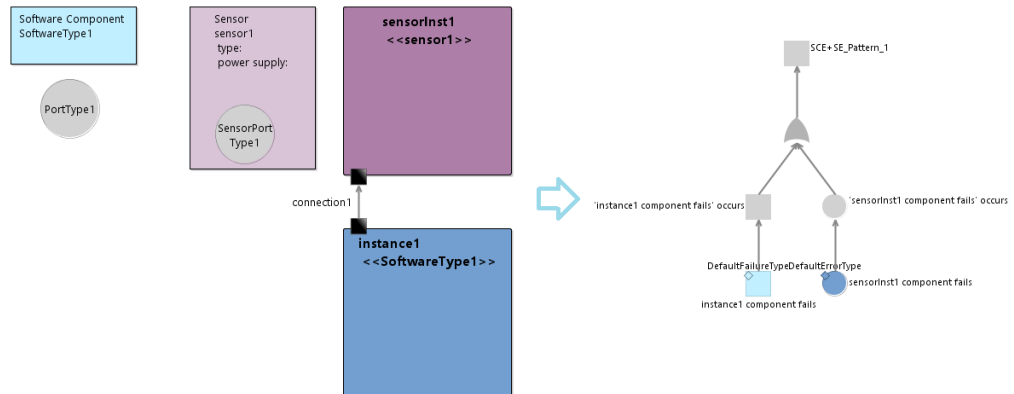


Figure 7.7. New connection between sensor and software component pattern

7.4.3 Connecting the Hazard to the Remaining Fault Tree

Since there are arbitrary connections and nestings of the component instances in the component diagram, there might be several partitions in the corresponding Fault Tree. Thus, it is necessary to connect all parts of the Fault Tree to the hazard. We create an AND gate as an input gate for the hazard event. This implies that the top hazard only occurs if all of its underlying components have failed. All events which correspond to a top level component instance and have no incoming connection will be linked to this very top AND gate. The semantics are, that all component instances with incoming connections are used by the connected components and because of this, if the component fails, the calling component will fail, too. In short, component instances with incoming connections will result in a failure or an error event with an outgoing connection in the Fault Tree and vice versa.

7.5 Sequence Diagram to LQN

Author: Jonas Scheurich

The transformation between sequence diagram and LQN is implemented only to the LQN direction. This transformation is an unidirectional transformation, but the implementation of the back direction LQN to sequence diagram don't improve the proof of concept of the CoWolf project, because the transformation items are in a one-to-one relationship for witches we already proved bidirectional in 7.2. Thus way we describe only the implemented direction.

We implemented the transformation sequence diagram to LQN based on the description of [CDMI11]. Cortellessa et. al. present a transformation that performs a three to one transformation: Activity diagrams, component diagrams and sequence diagrams to LQN.

The CoWolf co-evolution framework only supports one to one transformations at the moment (see Chapter 9.5). We changed the transformation of [CDMI11] to a sequence diagram to LQN transformation, described in the following sections.

7.5.1 Initial Preparation of the LQN Model

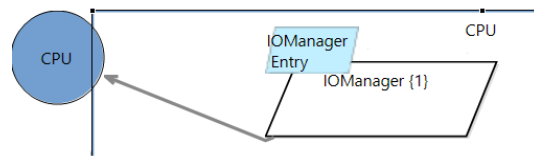


Figure 7.8. All created tasks are related to the CPU processor.

A sequence diagram supports no information about the machine the software is running on. Sequence diagrams are class based. So we assume, that the classes in a sequence diagram are running on a CPU and associate all tasks, mapped from a lifeline in a sequence diagram, to a CPU processor in the target LQN. In the first step of every sequence diagram to LQN transformation CoWolf checks if a CPU processor is part of the model and creates new processor named "CPU" if it is missing as shown in Figure 7.8.

7.5.2 Sequence Diagram: Lifelines

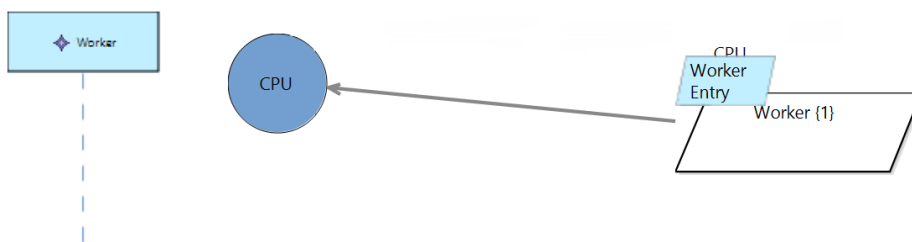


Figure 7.9. Each lifeline will be transformed to a task.

Every lifeline in a sequence diagram is mapped to a new task and a new entry type in a LQN, shown in Figure Section 7.9. The new task is associated to the default processor created in 7.5.1. CoWolf allows the user to change the processor of a task without influencing further co-evolutions.

7. Transformations

7.5.3 Sequence Diagram: Synchronous Messages

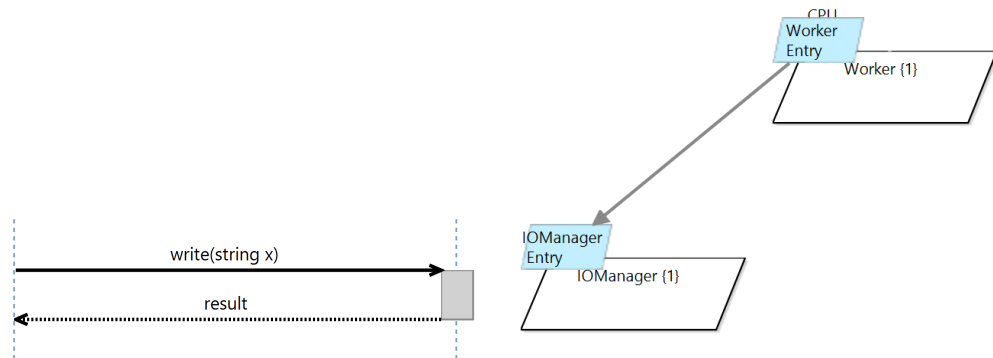


Figure 7.10. All synchronous messages between two lifelines (unidirected) are transformed to one synchronous call in a LQN model.

All synchronous messages directed from a lifeline l_1 to a lifeline l_2 in a sequence diagram are mapped to exactly one synchronous call in LQN (see Figure 7.10). The source task of the call is the task mapped from lifeline l_1 , the target task of the synchronous call is the task corresponding to lifeline l_2 .

7.5.4 Sequence Diagram: Asynchronous Messages

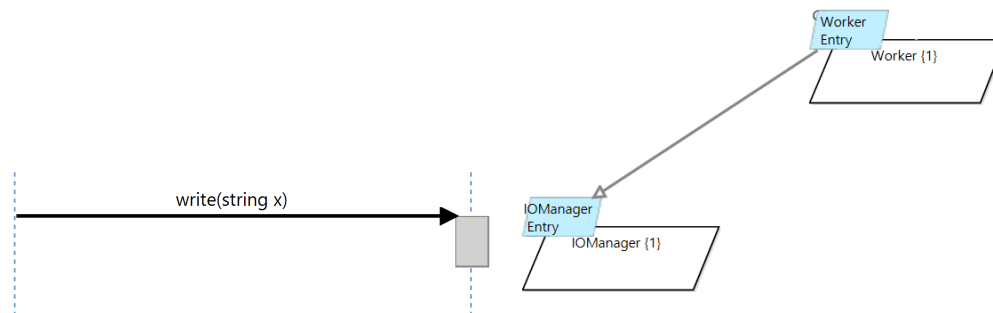


Figure 7.11. All asynchronous messages between two lifelines (on direction) will be transformed to one asynchronous call in a LQN model.

All asynchronous messages directed from a lifeline l_1 to a lifeline l_2 in a sequence diagram are mapped to exactly one asynchronous call in LQN (see Figure 7.11). The source task of the call is the task mapped from lifeline l_1 , the target task of the asynchronous call is the

7.5. Sequence Diagram to LQN

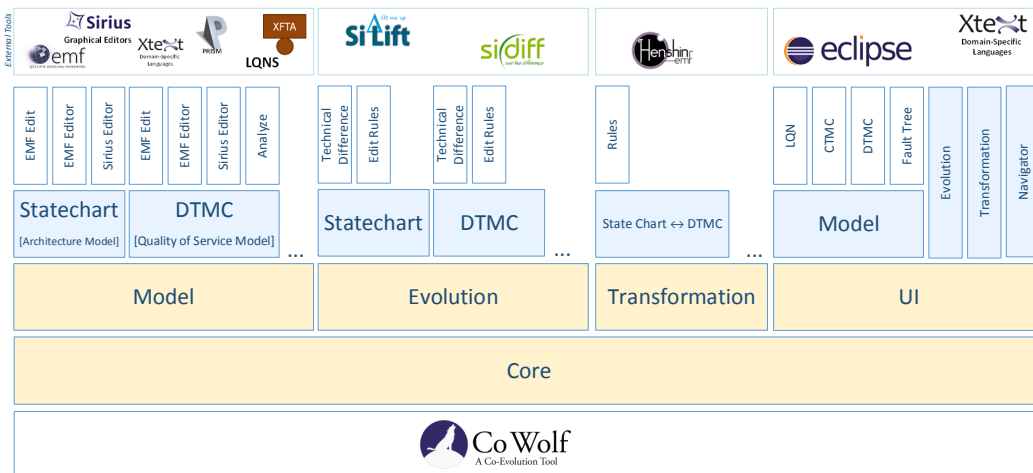
task corresponding to lifeline l_2 .

Architecture

Co Wolf consists mainly of five parts: Model, evolution, transformation, UI and core. What they contain and how they play together is explained in the following.

8.1 Concept and Overview

Author: Jonas Scheurich, Michael Müller



Overview of the components. The main-components (orange filled) of Co Wolf: Core, Model, Evolution and Transformation. The components of the main-components (blue filled) e.g. Statechart.

Figure 8.1. Overview of the components.

Co Wolf supports a sample of plugins for the eclipse development environment. Picture 8.1 shows an overview of the Co Wolf-architecture with the main-components Model, Evolution, Transformation and UI with the containing components (e.g. Statechart). Every component contains some sub-components with certain functionalities. All main-components, components and sub-components are represented by an eclipse plugin.

8. Architecture

The main-component *Model* (see Section 8.2.2, 8.2.3) contains a component for every supported model (see Chapter 6). A model component (e.g. Statechart) provides the meta model, the model implementation, the EMF-editor and a graphical editor. For quality of service models this component contains the analysis, too. The main-component Model is independent from the main-components evolution or transformation.

The evolution component contains for an associated model the evolution to calculate the difference between two versions of the associated model type.

The main-component *Transformation* contains components for every supported co-evolution type. A Transformation component supports a Co-Evolution in one or both directions of two associated models. In the CoWolf-project the transformations are realized with henshin (see Section 5.1.4). A Transformation component depends on to the two evolution components of the two associated models.

The main-component *Core* envelopes functionality that is used widely in other components. This comprises mainly utilities, e.g. a command line execution infrastructure and a version and association management system, as described in Section 9.2.

The main-component UI contains the user interface components for the evolution, the transformation, the CoWolf-perspective and some models.

8.2 Models

Author: Manuel Borja

The *Models* component contains all the logic related with the creation and edition of architectural and QoS models. In the case of QoS models this component is also responsible of the model analysis execution, which among others includes the set of activities to call the external analysis tool.

In order to guarantee cohesion and scalability, the logic of every model resides in a different component. Thus, we have eight different components which inherit from the components in the abstraction layer.

8.2.1 Abstraction Layer

The abstraction layer contains the common logic of the CoWolf component models. It contains as well the common structure of the supported models. Services like automatic ID creation for model elements and validation if a model is supported or not are included in its logic. The figure presents the structure of the abstraction layer

CommonBase

The CommonBase component contains the basic structure of every model supported by CoWolf. It is also responsible of the automatic ID-element creation, i.e. every time that a model's element is added a new unique not-visible-to-user identifier is created and assigned to its ID attribute. The automatic ID creation is in general triggered every time that a model file is saved. This is a necessary and important functionality in order to be capable of identify how the elements of two co-evolved models are related [see transformations].

Every model supported by CoWolf has to have the following attributes

- ▷ ID: unique element's identifier
- ▷ Name: name of the element

ModelManager

The ModelManager contains the abstract logic of every architectural and QoS model manager. In general a ModelManager is responsible of the model's naming and instantiation. In the case of QoS models the ModelManager contains also the analysis functionality.

ModelRegistry

In the ModelRegistry component resides the list of supported models. It is responsible, among others, of the instantiation of model managers

8.2.2 Architectural Models

This component contains the structure and logic of every architectural model. The classes of this component - which are generated with EMF - mirror the structure of the model and contains the logic related with creation and validation of model's elements.

The components in this layer are:

- ▷ Activity diagram
- ▷ Component diagram
- ▷ Sequence diagram
- ▷ State machine diagram

8. Architecture

8.2.3 Quality of Service Models

This component contains the structure and logic of every quality of service model. Most of the classes of this component are generated with EMF too and mirror the structure of every model as well.

The components in this layer are:

- ▷ CTMC diagram
- ▷ DTMC diagram
- ▷ Fault tree diagram
- ▷ LQN diagram

Furthermore every component is responsible of the preparation and invocation of the respective external tool in order to perform the corresponding quality analysis. In general the analysis process consists of:

- ▷ Translation: the EMF model is converted to a file in a format suitable for the analysis tool. This file, which is generated with help of Xtend, is saved into a temporary file.
- ▷ Script generation: an execution configuration file is generated. It contains the analysis parameters and the goal of the analysis, i.e. what is desired to be evaluated. In the case of LQN this information has to be present in the same model file created in the step 1.
- ▷ Invocation: the program is run passing as parameters the model and the analysis parameters.
- ▷ Read and parse results: the component read the results, which normally resides in a temporary file. The standard an error output console values are read as well. The results are iteratively read and transformed to an internal structure.
- ▷ Write results: The results are written into a html file.

8.3 Evolution

Author: Michael Zimmermann

CoWolf makes use of the SiDiff (see chap. 5.1.5) and SiLift (see chap. 5.1.6) functionalities to allow the user to compare different versions of a model, based on low-level changes but also as semantically lifted high-level changes. For this purpose, SiDiff calculates the correspondences as well as the technical low-level changes between the different model versions. SiLift takes this low-level changes and groups them into so called semantic change sets, a higher abstraction level representing user edit operations.

To enable the evolution feature of CoWolf for a specific model, besides SiDiff and SiLift mainly three elements are needed (see Figure 8.2):

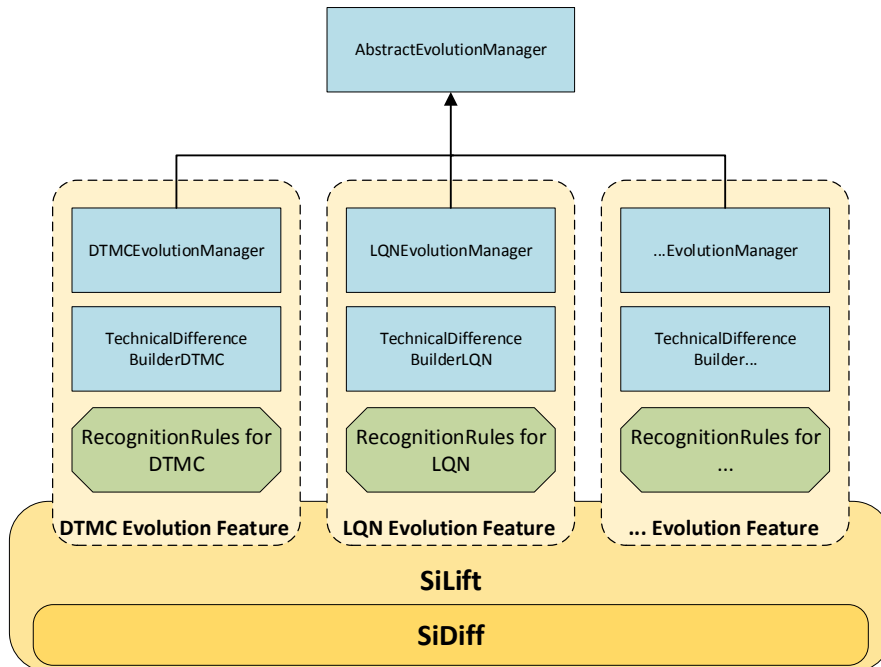


Figure 8.2. Important elements for the realization of the evolution feature of CoWolf. Blue boxes represent java classes, green octagons represent Henshin files and yellow/orange rounded boxes represent plug-ins.

- ▷ The model-specific `EvolutionManager` inherits from the `AbstractEvolutionManager` and specifies if the evolution feature for this model should be enabled. Here, the matcher that should be used to calculate the correspondences for this model is specified, too.
- ▷ The also model-specific `TechnicalDifferenceBuilder`: `SiDiff` needs this class in order to build the low-level difference. Here the desired model must be specified. Furthermore, model elements can be defined here that should be filtered and thus aren't part of the resulting low-level difference.
- ▷ The `SiLift-RuleBaseProject` contains all the recognition rules for the specific model. They enable the possibility of `SiLift` to detect user edit operations. The recognition rules are Henshin rules (see chap. 5.1.4) and are automatically generated from `SiLift` if the associated edit rules are located in the same plug-in.

8. Architecture

8.4 Co-Evolution

Author: Michael Müller

The *Co-Evolution* component (see [img. 8.1](#)) of *CoWolf* provides all the logic related to transformations between different model types. As already explained for the model and evolution components, each transformation resides in a separate Eclipse plugin.

8.4.1 Abstraction Layer

The abstraction layer of the Co-Evolution component contains the interface to Henshin (see [chap. 5.1.4](#)), which is used to execute transformation rules, as well as the common logic needed to perform the transformation of models. The main class that handles both mentioned items is the `AbstractTransformationManager` which has to be extended by transformation plugins.

AbstractTransformationManager

The `AbstractTransformationManager` provides the main transformation framework used in *CoWolf*. Before the transformation is invoked, the `AbstractTransformationManager` performs the following tasks:

- ▷ Loading of correct versions of models by using the version management of *CoWolf* (see [chap. 9.2](#))
- ▷ Performing evolution on source models by using the Evolution component of *CoWolf* (see [chap. 8.3](#))
- ▷ Loading of extensions which registered at its defined extension points (cf. [12.3.2](#), [12.3.2](#))

After successfully completing these tasks, the real transformation starts, where the developer can basically choose between two transformation strategies:

- ▷ Difference based transformation (see [chap. 12.3.2](#))
- ▷ Mapping transformation (see [chap. 12.3.2](#))

The `AbstractTransformationManager` executes the transformation based on the strategy chosen and returns the new result model instance.

8.4.2 Implementation of a Co-Evolution

Essentially it is necessary to create a `TransformationManager` which inherits from the `AbstractTransformationManager` and specifies, which two model classes this transformation can be handled. This process is described in detail in [Section 12.3.1](#).

In general it is sufficient to extend the `AbstractTransformationManager`, implement the abstract methods and provide Henshin rules for the difference based transformation or provide Henshin rules plus a mapping for the mapping based transformation.

8.5 Graphical Interface

Author: Verena Käfer

Co Wolf includes two possible user interfaces for every model. The standard EMF tree view and an individually created drag-and-drop view created with Sirius [Sir].

8.5.1 Tree View

Each tree view has its own plugin. This plugin can be easily created from an existing generator model. Therefore each plug-in has the same elements:

- ▷ An action bar contributor
- ▷ An editor to change modelled elements
- ▷ A singleton editor plugin class as activator
- ▷ A wizard to create new models

All four classes are connected in the plugin.xml, either in extension points or as activator.

8.5.2 Sirius

The architecture of a Sirius editor plugin is even simpler. The main part is an .odesign file which includes every information needed to create and open a graphical representation of a model. Additionally there may be service classes which include more complex algorithms to create or edit diagram elements. At last there is also an activator class.

The .odesign File

This file is the most complex part of a Sirius-based editor. It includes mostly two parts:

- ▷ The mappings which map a diagram element to a model element under given conditions. This is only the representation, that describes which existing model element will be represented graphically in what way.
- ▷ The create and edit functions. This part describes the conditions for creating new elements or editing existing ones and what must be done in the underlying model.

All parts are combined in a viewpoint which declares on which model types this graphical editor can work.

Implementation

In the previous chapters was explained how the theories behind CoWolf work. How the models work and how the transformations are done. This chapter describes how CoWolf was implemented. The graphical editors, the analysis and the transformations.

9.1 Models

There are two different ways how models can be edited in CoWolf: In a tree view and in a graphical editor. How these editors are implemented is shown in the following section.

9.1.1 Textual Editor

Author: Johannes Wolf

For the textual presentation and editing of the models we use the tree view editor of EMF (see Section 5.1). With EMF you can create the code for the tree view editor automatically by defining a generator model. The generator model can also be created automatically by using the EMF wizard. After creating the generator model, the developer can customize the allowed user actions. It is for example possible to enable or disable the opportunity to create a child of a particular element. After that, the code creation of the textual editor can be started. Figure 9.1 shows the EMF tree view editor for the DTMC model. On the left side you can see the model instance as a tree and edit the model by using the context menu. On the right side is the properties view, where you can edit the properties of the selected element.

9.1.2 Graphical Editor

Author: Johannes Wolf and Verena Käfer

As described in Section 8.5.2 Sirius uses an *.odesign* file to define the graphical editor for the meta models. In this section we will look more in detail how the graphical mapping and editing of the models are defined.

9. Implementation

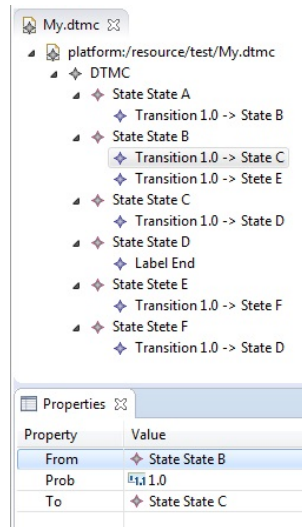


Figure 9.1. User interface of the EMF tree view editor

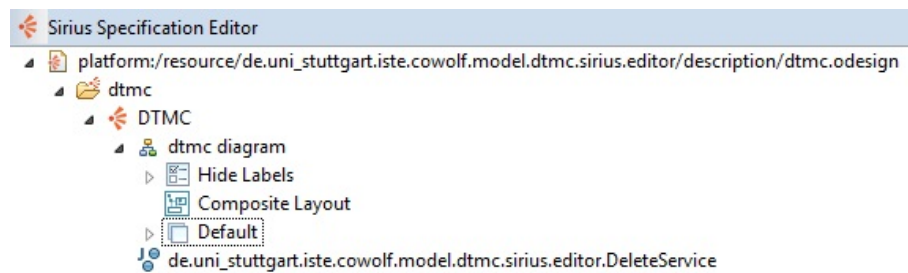


Figure 9.2. Basic structure of the .odesign file

Graphical Mapping

For each meta model you have to create a viewpoint first. It is possible to provide many viewpoints for the same meta model which focus on different aspects of the model. In our case we provide one viewpoint for each meta model which is a node-link diagram that displays all elements of the metamodel.

The second step is to create a layer that contains the graphical mapping. You can define many layers, which gives the opportunity to hide and show different elements of the model. It is also possible to define filters for this purpose. In Figure 9.2 you can see the basic structure of the *.odesign* file.

Sirius provides a set of *Diagram Elements* which represent the mapping of a meta model element to its graphical representation. The graphical representation is described by a *Style*.

You can choose mainly between basic shapes or use your own images for the representation. Sirius provides the opportunity to define *Conditional Styles* to show different graphical representations for the same Ecore class depending on a condition expression. For example this is used to show the initial state of the CTMC model with a thicker border than the normal states. To define the expression it is possible to use OCL, Aceleo or an own interpreter defined by Sirius. You can also call Java functions instead. The following list shows the *Diagram Elements* that we used to map our meta models to a node-link diagram:

- ▷ *Node*: A node maps a single Ecore class of the meta model to a graphical element. Nodes are for example used to map the states of the DTMC model to a circle.
- ▷ *Container*: A container also maps an Ecore class to a graphical element but it can additionally contain other *Diagram Elements* of any kind. For example a container is used to display composite states of the Statechart model. The composite state can contain many states and composite states as well. To support the nesting of many composite states you have to enable the inheritance of the mappings from the ancestors.
- ▷ *Bordered Node*: The bordered node is a node which is clipped to the border of another node. This node is used in the DTMC model to display labels or in the component diagram to display ports of a component.

Each of these types can be extended by a label which contains a text like the name or a dynamic expression. For the edge mapping between the nodes or containers there are two types:

- ▷ *Relation Based Edge*: This type of edge mapping can be used, if an Ecore class refers another Ecore class. You also have to define a target finder expression to get the semantic element you want to connect. In simple meta models this expression is a attribute of the Ecore class.
- ▷ *Element Based Edge*: This edge mapping is used, if the connection between two elements is represented by an Ecore class and not by a single reference. For example this is the case in the Ecore class *Transition* of the DTMC meta model. Beside the target finder expression you also have to define the source finder expression do get the semantic source element of the edge.

In Figure 9.3 you can see an example of the mapping elements of the Statechart model.

Model Editing

To enable the user to edit elements of the model you have to define one or many tool sections. This sections group the supported tools in the user view (see Figure 9.4).

The following list shows the user actions which are mainly supported in the graphical editor:

9. Implementation

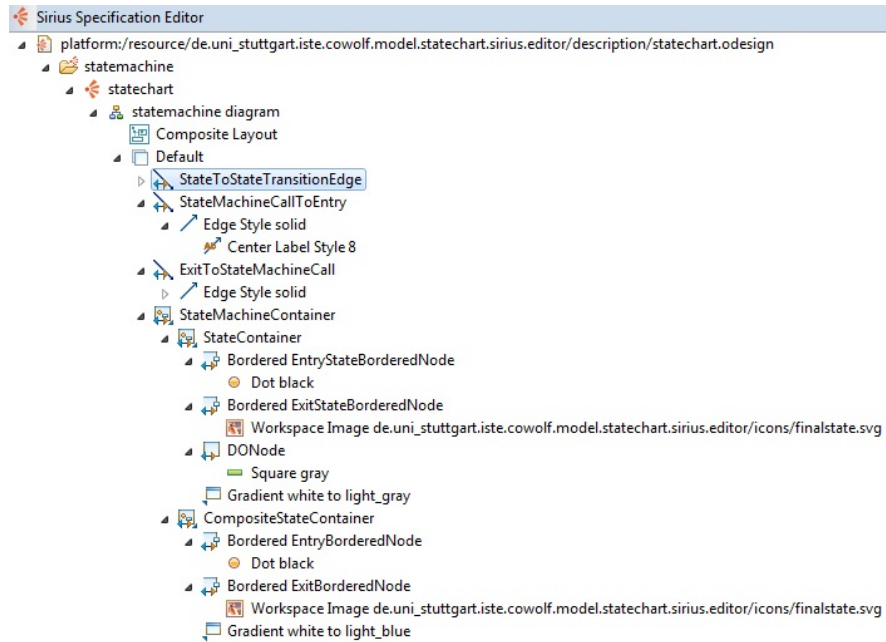


Figure 9.3. Structure of the .odesign file for the Statechart model

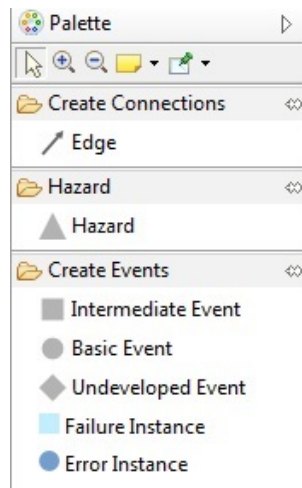


Figure 9.4. User view of the edit palette for the fault tree model editor

▷ *Element Creation*

- ▷ *Node/Container Creation:* For the creation of a node or a container you have to define the mapping between the creation tool and the diagram element you want to create. You also have to define the reference name of the parent element. If it is possible to create an element within different parent elements (like container, node or bordered node) you have to define a conditional mapping for the reference (see Figure 9.5).
- ▷ *Edge Creation:* The edge creation works the same as the node creation. Sirius additionally provides the source and target element as a variable to set the edge mapping.

▷ *Element Editing*

Beside the element creation the user has the opportunity to edit existing elements.

- ▷ *Delete Element:* By default the element deletion is enabled by selecting a graphical representation and using the context menu or pressing the *del* key. In more complex meta models it is necessary to customize the element deletion. One reason could be that you also have to delete the parent elements which contains the selected element. This is for example the case in the LQN model. In other cases its necessary to prevent an element deletion and set the element to delete manually. This is the case if you select an edge which semantically represents the source node.
- ▷ *Direct Label Edit:* The direct label edit enables the user to set the value of a property by editing the label of the element.
- ▷ *Reconnect Edge:* In graphical editors of simple meta models like DTMC, the edge reconnection is supported. Here it have to be defined how the new source or the new target can be found by using an expression.

In other cases also user actions like a selection wizard or drag and drop are supported. To prevent some wired behavior when copying and pasting graphical elements, this function was disabled in some editors for the user.

Specialities for Sequence Diagrams

Sirius has special elements for Sequence Diagrams. As sequence diagrams are very complex diagrams, we decided to do our graphical editor based on the UML-Designer tutorial [Umlb] which already has sequence diagram support. Therefore we also have or meta model based on the original EMF UML meta model [Umla]. What elements are included in our meta model can be seen in Section 6.1.3.

Specialities for LQNs

As we use the LQN Solver to analyse LQN models (See Section 9.3.3), we took the meta model from Palladio [Pal]. Palladio also uses the LQN Solver to analyse their model and its LQN meta model has the necessary formality.

9. Implementation

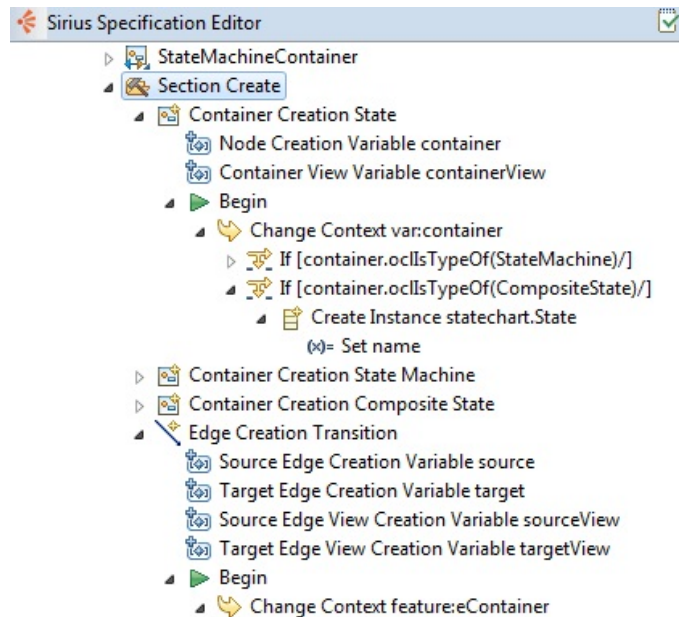


Figure 9.5. Definition of the element creation in Sirius

Problems

The work with Sirius wasn't always easy. As much as it is a good possibility to create own graphical editors, it also has some bugs. During the development some bugs occurred that were either already known or confirmed by the Sirius development team. Some of the bugs got a workaround which makes them invisible to the user but some were not that easy to fix.

The biggest problems occurred with the LQN meta model. As we took the LQN meta model from Palladio, we did not create it ourselves. After the first problems occurred we investigated and saw that there are two different possibilities to create Ecore meta models: The way we used with the EMF model editor and importing a XML schema. Sirius has problems with the XML schema version of the meta models and we strongly believe that Palladio used this method to create the model, especially as some problems only occur with the LQN models.

The one special LQN problem is that the according model instance is not stored in the XML code of the representation file at all. That means that when the representation file is created, the file can be opened as the session object holds all needed information. But when the session is closed, like after a restart of Eclipse, this information is gone and the editor shows an error. Unfortunately it was not possible to create a workaround for this

error.

An other bug we found is that Sirius takes an URL to the according model instance when a new representation is created. Unfortunately when the file is in a sub-folder, the whole path is stored in the XML file. That would make sense, if Sirius would not try to load the file with the given path relative the path of the representation file. That means that it is not possible to have the representation file in a different folder than the model instance and also that we needed a workaround. No fix the wrong path in the XML file we replaced it manually with the file name only.

The third problem we had was the moving and renaming of model instance files. The idea was basically to reset the connection to the new model instance file. But using the official way led to the problem, that after a reset the whole user-specific layout was gone, as Sirius cannot decide if the same elements are available or if a complete new file was used. Therefore we decided to manually reset the path to the file directly in the XML code. So far so good, now a new problem occurred. Sirius has a listener on resource changes. When a file is renamed or moved the representation file gets invalid and instead of changing the representation file they just delete the reference to the file. Very clever. Now we have a workaround for two possibilities: When we are faster than the Sirius listener, we reset the reference to the file and everything is fine. When the Sirius listener is faster, we set the reference to the changed file, but unfortunately in this case the user-specific layout is lost.

9.2 Version Management

Author: Philipp Niethammer

The co-evolution between models is based on knowledge about changes in the source model between the last co-evolution and the current Model Version. Therefore, CoWolf provides an integrated version system for all supported models that automatically tracks changes in the models and creates versions on special occasions. Additionally, this system is also capable to manage associations between different Model Instances. At the moment, associations are used to mark former transformations between model instances. On these grounds, the component for this functionality is called `ModelAssociationManager`. As it is an integral part of the framework, it is part of the Core plugin.

The CoWolf version management is not supposed to replace a common VCS like Git, Subversion, etc., but to meet the special requirements of model evolution and co-evolution. As a consequence, it is designed to work in parallel to other VCS environments.

9.2.1 Model and Association Management

The `ModelAssociationManager` is based on an Ecore Model with an Model Instance for each Eclipse project. For the sake of clarity, we will refer to a `ModelAssociationManager` model's

9. Implementation

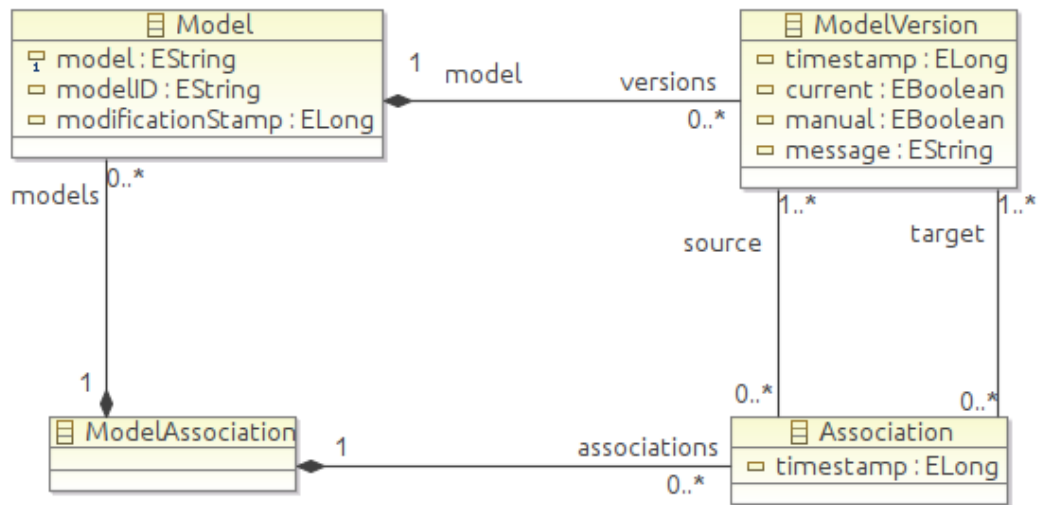


Figure 9.6. ModelAssociationManager Model

instance as manager.

The manager contains information for each instance of supported models in the project. Beside the project relative path of the model instance and the ID of the model's root object, if available, it also contains a modification stamp to easily identify if the file in the workspace was changed since the last visit, and information about each version of the instance. A version is identified by the Unix time stamp of its creation and can have an optional message.

Additionally, the manager holds associations. An association is a directed n:m relationship between Model Versions. They are augmented by the date of creation and can save Henshin traces that evolved from the transformation they describe. Figure 9.6 shows a class diagram of the ModelAssociationManager's model.

The manager data is saved serialized as XML in a hidden file named `.modelassociation` in the Eclipse project root. The data is automatically loaded from the file system on demand and saved immediately after changes in the data to prevent data losses on an Eclipse crash.

To synchronize operations in the Eclipse workspace with the manager, it is monitored. Thereby we are able to update the manager if a Model Instance is created, moved, renamed or deleted. Additionally, if a managed file is moved to another project, the information about the Model Instance and all its versions are moved to the target project's manager. Unfortunately, as Eclipse isn't able to detect the copying of a file [UI08], we can't duplicate the versions in this case. It is therefore handled in the same way as the creation of a Model Instance.

9.2.2 Model Version Management

As mentioned above, it is not within the scope of this project to provide a complete VCS. However, we need easy access to Model Versions used in former transformations to identify the interim changes.

Thus, we decided for a very basic file based system. Therefore, each Eclipse project contains the hidden folder `.modelversions`. In this folder for each Model Instance a folder structure is created, resembling the whole project relative path of the file. As described, a version is identified by a UNIX timestamp in the manager. This timestamp is now used as filename of the version file, which simply is the copy of the Model Instance state at the given time. For example, a version of the Model Instance `model/main.dtmc` is saved as `.modelversions/model/main.dtmc/3126447571.version`. In this way, a version can simply be read by services provided by the respective Model Plugin.

Versions are automatically created on special occasions if it either might be needed for later co-evolutions or to backup manual changes prior to automatic modification. In detail, that comprises:

- ▷ On creation of the model instance. This saves an empty instance, only containing the ID of the root object. It is used for the first transformation.
- ▷ The source model instance prior transformation, if it differs from the latest version. This version is needed in future for the next co-evolution.
- ▷ The target model instance prior transformation, if it differs from the latest version, to backup manual changes.
- ▷ The target model instance after transformation. After transformation, the Henshin traces reference on this model. It is therefore needed in future to resolve these references. Additionally, this version is part of the association between source and target model instance, formed by the transformation.

All Model Versions are automatically removed with the deletion of the Model Instance. Further more the user is able to create versions manually at any time. These versions are marked as "manual" in the manager.

9.3 Analysis

The analysis of the Quality of Service (QoS) models is done in external tools, but the customization and the result view are created by *Co Wolf* to support high usability. However, most external tools use different model languages than defined in this project, so they need to be transformed into the other format. Furthermore the parameters of the analysis have to be defined by the user. In the following sections the implementation of the analysis for DTMCs, CTMCs, LQNs and Fault trees are explained.

9. Implementation

9.3.1 Analysis of a DTMC Model

Author: David Steinhart

DTMC models can be analyzed using the PRISM model checker (5.2.1). *Co Wolf* provides reliability validation using reachability analysis. For each state and each label in the DTMC, for example error states, the reachability can be analyzed. This helps the user to decide whether he should change the software architecture or make a component more robust to failure.

9.3.2 Analysis of a CTMC Model

Author: David Steinhart

Similar to DTMC models, CTMC models are analyzed using the PRISM model checker (5.2.1). CTMC models are used for performance and reliability analyses. Reliability can be validated using the reachability of critical states, for example error states. Performance can be validated in multiple ways. *Co Wolf* provides a wizard (see 9.7) which helps to create default properties, which are "Steady State Probability", "Probabilistic Response", "Probabilistic Until" and "Probabilistic Existence".

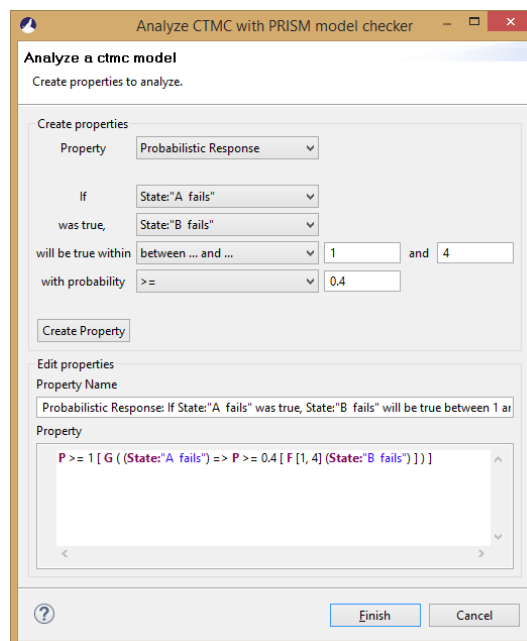


Figure 9.7. CTMC properties wizard

"Steady State Probability" calculates the probability that condition A will eventually become true. "Probabilistic Response" calculates the probability that condition B will always become true in a timeframe after condition A was true. "Probabilistic Until" checks if condition

A always was true before condition B becomes true. "Probabilistic Existence" checks if a condition becomes true in a timeframe. As there are many more possibilities to evaluate CTMC models, additional properties can be created and edited in an Xtext-based text editor (5.1.2).

9.3.3 Analysis of an LQN Model

Author: Manuel Borja

The analysis of an LQN model is performed by the LQN Solver (see Section 5.2.3) developed by the University of Carleton. In order to solve an LQN model with LQN Solver, is necessary to transform the EMF model into a *.lqn* file. This transformation is made with XTend. A snippet of the *.lqn* file's template is shown in the Figure 9.8.

```

11  # Processor definitions
12  P 0
13  p <proc-id> <sched> <opt-mult> <opt-repl> <opt-obs>
14  -1
15
16  # Group definitions
17  U 0
18  g <group-id> <real> <opt-cap> <proc-id>
19  -1
20
21  # Task definitions
22  T 0
23  t <task-id> <sched> <entry-list> -1 <proc-id> <opt-pri> <opt-think-time>
24  <opt-mult> <opt-repl> <opt-grp> <opt-obs>
25  -1
26
27  s # Entry definitions
28  E 0
29  A <activity-id>
30  s <entry-id> <real> ... -1 <opt-obs>
31  y <entry-id> <entry-id> <real> ... -1 <opt-obs>
32  -1
33
34  # Activity definitions
35  A <task-id>
36  s <activity-id> <real> <opt-obs>
37  y <activity-id> <entry-id> <real> <opt-obs>

```

Figure 9.8. Snippet of the *.lqn* file's template

Solver Parameters

It is possible to configure a set of parameters on the activities in order to model the workload of a system. These parameters are:

- ▷ Execution demand: The time demand on the CPU or other device

9. Implementation

- ▷ Wait delay or think time: It can be used to model a delay which not involves the processor.
- ▷ Mean synchronous requests to another entry.
- ▷ Mean asynchronous requests to another entry.
- ▷ Probability of forwarding the request to another entry rather than reply it.

Results

CoWolf presents the following results associated with every element of the model.

- ▷ Processor statistics:
 - ▷ Utilization: Amount of processor's usage
- ▷ Task statistics:
 - ▷ Throughput: Throughput of the task
 - ▷ Processor's utilization: Amount of processor utilization by the task
- ▷ Entry statistics:
 - ▷ Throughput: Throughput of the entry
 - ▷ Processor's utilization: Amount of processor utilization by the entry
- ▷ Activity statistics:
 - ▷ Service time: Total time that the activity uses to process a request. The service time is calculated based on the delays of the following events:
 - ▷ Queuing for the processor
 - ▷ Service at the processor
 - ▷ Queuing for service tasks
 - ▷ Phase one service time at serving tasks
 - ▷ Service time variance: Variance of the service time.

9.3.4 Analysis of a Fault Tree Model

Author: Manuel Borja

The analysis of a Fault Tree model is performed by xFTA (see Section 5.2.2)

Probabilistic Calculations

- ▷ Probability of top event: As its name indicates, it is the probability that the hazard occurs. Additionally, xFTA calculates the importance factors and their values are shown by CoWolf as well. The importance factors indicates how much is the contribution of the different components to the overall risk.
- ▷ Minimal cutsets: A cutset is a combination of basic events which, when all of them occur, the hazard occurs as well. A cutset is minimal if no of it subsets is a cutset. The contribution of a minimal cutset is the probability of this cutset divided by the sum of the probabilities of the calculated minimal cutsets.

9.4 Evolution

Author: Michael Zimmermann

Since CoWolf uses SiDiff, SiLift and SERGe (see Chapter 5.1.5, 5.1.6 as well as 8.3) to realize the evolution feature, the main part of the implementation here was to integrate (respectively use, in case of SERGe) these externally developed plug-ins. The above mentioned tools offer great functionalities to calculate and present differences of model versions. But as these tools are also still under development the integration wasn't quite easy. Therefore, this chapter will give an overview how the final implementation of the evolution feature looks like as well as of some problems that occurred during the implementation of it.

9.4.1 EvolutionManager

Author: Michael Zimmermann

Each supported model of CoWolf has its own model-specific EvolutionManager that inherits from the AbstractEvolutionManager. Figure 9.9 shows the UML class diagram of the AbstractEvolutionManager. If not specified in the subclass, per default the EMF-Compare Matcher is used to find the correspondences between the model versions. By overriding the method getEvolutionTypeInfo() the matcher can be specified for the individual model plug-ins. The figure also shows the abstract method getManagedClass() which is implemented by the subclasses and returns the model class. This method is used in the isManaged() methods to decide if the evolution feature is supported for the specified model.

Also, most of the SiLift integration is done in the AbstractEvolutionManager. The getDefaultSetting() method for example creates a LiftingSettings object which is required from SiLift for the difference calculation. The class diagram also shows the two methods evolve() and getDiff() that internally call the SiLift liftMeUp(Resource, Resource, LiftingSettings) method which gets two model versions and the LiftingSettings and then computes the lifted difference of this two model versions. As you can see, the evolve() and the getDiff() methods return different objects. This is because SiLift provides two

9. Implementation

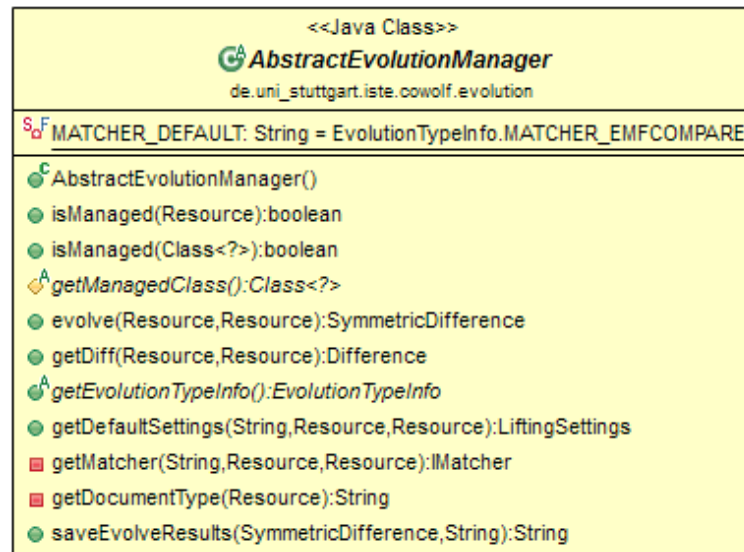


Figure 9.9. UML class diagram of the `AbstractEvolutionManager` class.

different `liftMeUp()` methods: one that returns a `SymmetricDifference` and a second that returns a `Difference` object. The `SymmetricDifference` object just contains the lifted difference of the two model versions. The `Difference` object however contains this lifted symmetric difference as well as an asymmetric difference, which is used for patching purposes (see chap. 13.4).

9.4.2 TechnicalDifferenceBuilder

Author: Michael Zimmermann

Besides its own `EvolutionManager`, each supported model of `CoWolf` also has its own `TechnicalDifferenceBuilder`. Figure 9.10 shows the abstract `TechnicalDifferenceBuilder` class. `SiDiff` needs this class in order to build the low-level difference. For example, the of the individual subclass supported model must be specified here. Furthermore, with the `getUnconsidered*Types()` methods, model elements are defined here that should be filtered and thus aren't part of the resulting low-level difference. If no elements should be filtered, the methods can just return an empty set. In the implementation of our plug-ins only `Henshin` elements are filtered, although actually no model instance should contain such elements. Mainly the filtering of `Henshin` elements is a leftover, since at the beginning `Henshin` traces (see chap. 5.1.4) were stored in the model instances.

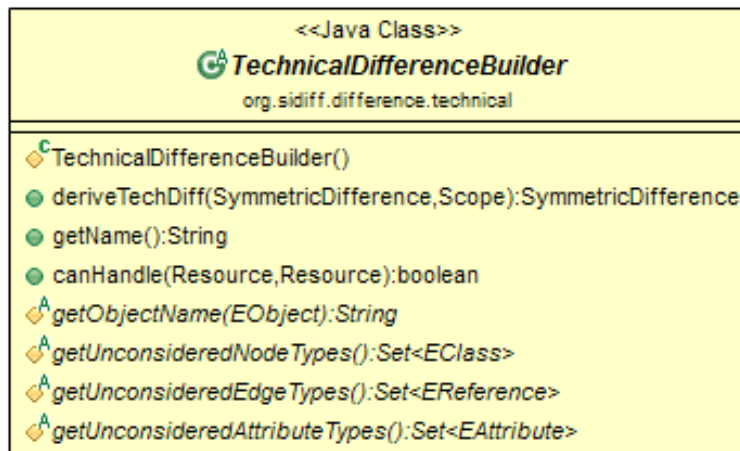


Figure 9.10. UML class diagram of the `TechnicalDifferenceBuilder` class (private attributes and operations are hidden).

9.4.3 SiLift-RuleBaseProject

Author: Michael Zimmermann

The third needed component for a model evolution feature is the `SiLift-RuleBaseProject`. Again, this is needed for all models supported by `CoWolf`. The `SiLift-RuleBaseProject` contains all recognition rules and a `rulebase` file (for managing the recognition rules) for the specific model. They are used by `SiLift` to detect user edit operations. The recognition rules as well as the `rulebase` file are automatically generated by `SiLift` during the workspace build process.

9.4.4 Problems

Author: Michael Zimmermann

As mentioned in Chapter 3, we used the continuous-delivery technique with a traffic light that indicates if after a push to the master branch the build process still was successful. The advantage of this technique among other things is the possibility to detect error-prone pushes quickly. But from time to time it happened that the build process failed not because of a bad push to the master but because something in `SiDiff` or `SiLift` was changed. As already mentioned, `SiDiff` and `SiLift` are also still under development and thus it's not surprising that their API isn't that stable and now and then some bigger changes are made. But since both tools don't provide something like a changelog on their project sites it was very time consuming to find and fix the problems every time again. Sometimes luckily just a new method needed to be implemented, but another time new plug-ins for all of our models were needed and old existing plug-ins became unnecessary. Therefore, to be independent of `SiLift` updates, we decided to create an own repository containing a `SiLift`

9. Implementation

version (1.0.0.201409111244) that was known to work well with CoWolf.

Also in an earlier version of SiLift every time the edit rules were changed, the recognition rules as well as the rulebase file (containing the recognition rules) needed to be created manually again. In a later version of SiLift this was automated and thus, saved a lot of time.

SERGe, the tool to generate the needed atomic edit rules also was changed from time to time. At the beginning of our implementation phase, SERGe needed its own Eclipse instance because the dependencies of SERGe weren't compatible with the plug-ins needed of our development Eclipse instance. In a later SERGe version this problem was fixed and SERGe could be installed on the development Eclipse instance. This helped a lot because now the model plug-ins for example no longer had to be synchronized on both Eclipse instances. But with the SERGe update also the scheme of the configuration file was changed. Thus, all SERGe configuration files of all of our models needed to be adapted and checked again if the edit rules generated based on them were still working.

But besides external tools, we also created problems ourselves. Our models weren't very stable from the beginning and needed to be updated occasionally. Thus, every time also the edit rules and hence the recognition rules as well as the rulebase file needed to be renewed.

9.4.5 SiLift Rulebase Maven Plugin

Author: Rene Trefft

SiLift generates the recognition rules and the rulebase file during the build of the Eclipse Workspace. As a general rule of thumb is to avoid storing generated files in a repository, a Maven plug-in was developed which performs the generation during the build of CoWolf.

The SiLift Rulebase Maven Plug-in provides two goals. The goal `register-ecore-model` looks for any Ecore files in a directory `model` in the root of a project and register them, more precisely said adds them to the EPackage Registry. The second goal `build` generates the recognition rules and builds the rulebase file by using the edit rules in the directory `editrules` relative to project root. Note, a project which has not the SiLift rulebase nature in its project file is skipped. Only recognition rules from valid edit rules (no error in the Eclipse workspace) will be generated. An invalid edit rule will be skipped and the error message of the validation printed.

As a requirement for the building is the availability of the appropriate Ecore Model, the Ecore model (and any dependent models) must be registered first. Thus, we have to ensure the following build order:

1. Common Base Model project:
`de.uni_stuttgart.iste.cowolf.model`
2. Project of model for which the rulebase should be built for:
`de.uni_stuttgart.iste.cowolf.model.<modelName>`
3. Edit rules project which contains the edit rules:
`de.uni_stuttgart.iste.cowolf.evolution.<modelName>.editrules`

The build order is simply given by the order of the projects / modules in the POM of the parent project `de.uni_stuttgart.iste.cowolf.parent`. The generated rulebase and the recognition rules will be stored in the edit rules project.

For the execution of the build goal some SiLift and SiDiff JARs are necessary. Note, the Maven plug-in is not a plug-in project as the CoWolf plug-ins, but a standard Java project, so Tycho (see chap. 3.5.3) isn't used. Consequently, the standard Maven dependency management applies which means any dependencies must be declared in the POM and their location must be a Maven repository (dependencies locally stored in the project is a not recommended alternative). As the necessary SiLift plug-ins are not available in such a repository, we stored them in our Nexus Maven repository on the Lismore server. It's referenced in the POM of the project.

The Maven plug-in is not added as a module in the parent project, because it is not directly a part of CoWolf and in addition changes occurring rarely. Instead, it can be built separately whenever it's necessary.

9.5 Co-Evolution Framework

The co-evolution framework is a central component of CoWolf. It is very prone to errors, though, as it uses many other components of the system. That includes model plugins for source and target, the evolution feature of the source model type, the version and association management system and hence EMF, SiLift and Henshin as external components. All these are combined to perform a single action, the execution of the co-evolution rules. It is plain that small faults in these components can simply propagate and manifest at this point. Then, again, due to the complexity of interactions it is often hard to track and deduce the problem.

This section provides an overview of selected parts of the co-evolution framework and problems we encountered in the implementation.

9.5.1 Resources

Author: Philipp Niethammer

The transformation requires a whole lot of information from different sources. This includes different model instances, traces between models and a change set. Fortunately, all these sources are defined using EMF and thus can be handled in the same way.

Each source is represented as EMF resource which contains information about the file system path and provides file system operations. The data is represented as objects of the base type `EObject` contained in the resource. These objects do a lot of work in the background, for example maintaining references within the resource and to other resources, or more specific, if a reference is added to one object, the opposite direction is automatically added to the target object. While this is very handy in normal usage, it leads

9. Implementation

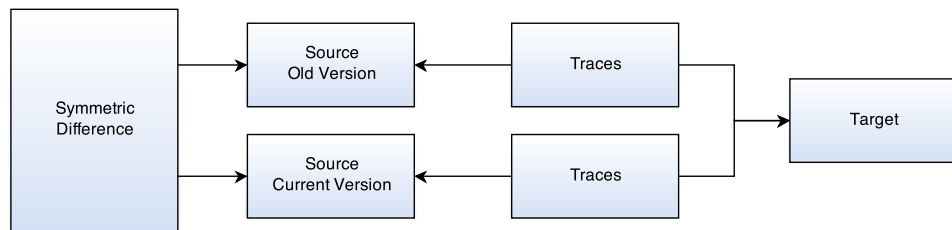


Figure 9.11. The wanted structure of a transformation graph.

to a very high coupling between objects and even resources that can lead to problematic and obscure side-effects.

To at least restrain side-effects to the co-evolution framework, as a first step we create independent copies of all resources before running any operation (even reading) on them. Additionally, we replace all file system URLs by pre-defined virtual identifiers to avoid confusions and separate the copy even stricter. We create an EMF resource set that contains the independent resources and provides easy access later on. A resource set is said to resolve references between resources using the registered resources [IBM+08], we are not really sure of it, as its behavior does not always support this.

The resource set contains the following resources by their virtual identifier. In source code, we are using constant URL fields or methods to access them.

- ▷ `transform:old`: The old source model version.
- ▷ `transform:source`: The current source model version.
- ▷ `transform:target`: The target model instance.
- ▷ `transform:oldtraces`: The traces leading from the old source model to the target model.
- ▷ `transform:traces`: The traces leading from the current source model to the target model and the resulting traces.
- ▷ `transform:diff`: The SiLift differences.
- ▷ `transform:result`: The resulting target model.

9.5.2 Transformation Graph

Author: Philipp Niethammer

Instead of resources or resource sets, Henshin is working on an EGraph object. The graph consists of nodes holding EObjects and edges describing a reference between two objects.

In theory and according to documentation, it should be trivial to add all resources to this graph, since it resolves dependent resources when a resource is added (using the resource set). Simply adding each resource should do the business and result in a graph

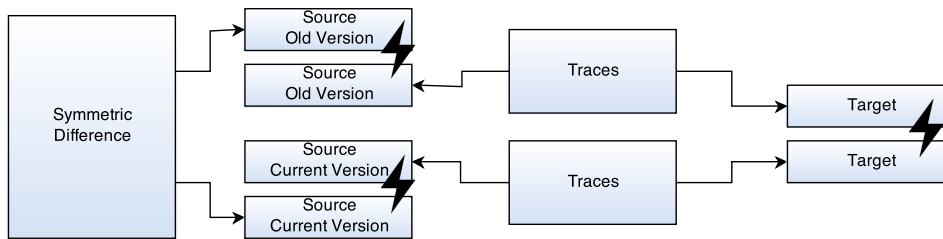


Figure 9.12. The graph structure after adding all resources.

as drafted in Figure 9.11. In practice, however, this led to duplicates of some resources in special occasions. For example, the differences referenced to another copy of the source model instance than the traces and old and current traces referenced to their own target model instance, as in Figure 9.12

We found that this can be solved by only adding specific resources in a specific order to the graph. Thus, we created the method `generateGraph` to generate the graph using the following rules:

1. Add `transform:diff` set to the graph.
2. If there are no traces in `transform:traces`, add `transform:source` to the graph. Else, add `transform:traces`.
3. If and only if there are traces in `transform:tracesOld`, add it to the graph. Else, do nothing (*in contrast to `transform:traces`!*).
4. If and only if both trace resources are empty, add `transform:target` to the graph.

The idea behind this is to build the graph only using resources that are not already referenced by other resources in the graph. If there is any trace, we conclude that the target model instance is already part of the graph, thus we won't add it again. We can't explain, however, why for example the source resources are not added twice in this way, because they are referenced by both, the differences and the traces.

9.5.3 Traces

Author: Philipp Niethammer

Henshin traces are a very important component of the transformation framework. They define correlations between source and target model and are therefore essential for the co-evolution process. A trace is a n:m relationship between different EObjects and has an optional name attribute. Additionally, traces can contain subtraces and form a hierarchical structure in this way. Although, subtraces are not supported by `CoWolf` at the moment as explained below.

9. Implementation

Usually, the traces are saved in the target resource. But as traces are not directed, e.g., the State Chart - DTMC transformation (Section 7.1) uses the same traces as the DTMC - State Chart transformation, we decided to save them separately from the model instances to have easy access from either side and to keep the model instances small as well. Additionally, we need to keep the traces of every former co-evolution which would make the storage in a model resource even more complex. To avoid problems with file system actions like renaming model instances, we also decided to keep the references purely virtual. That means, they are always referencing to `transform:source` and `transform:target` respectively.

As we always work with virtual references, they should be easily resolved to the corresponding resources when added to the resource set. Unfortunately, we encountered several problems: First, we needed traces resolved against both, the old and the current source model version. Second, we had some strange behavior with resolving the sources to even one of both. We were able to resolve traces and sources and print out the relations between them but once added to the graph, they didn't seem to be resolved at all. Third, even more obscure was a behavior we observed on the target side in special occasions: Being correctly resolved in the resource set, even with both sets of traces linking to the same target objects, after adding it to the graph the linked object were not the same as before. In fact, they were an earlier version of the target model instance. We do not have any clue where this version came from in the situation.

To solve these problems, we wrote a special method `resolveTraceSource` that reads each incoming trace, looks up the virtual URLs of all referenced objects in the resource set and creates a new trace that references to these objects. This solved the problems listed above and worked fine for all tests we performed. However, as the reasons for the described misbehavior are absolutely unclear to us, we can under no circumstances guarantee that there is no scenario in which this is not working correctly. Additionally, the method only processes direct traces in the list and doesn't copy any subtraces. We don't use subtraces up to now and thus saw no need in investing time into it in respect to the complexity of this topic. It should though be possible to add support for subtraces by calling the method recursively.

9.5.4 Rule Mapping

Author: Michael Müller, Rene Trefft

The rule mapping consists of mappings between SiLift specific `ChangeSets` or `Changes` of the `SymmetricDifference` and the Henshin rules used by the transformation to propagate the changes onto the model to co-evolve. It is implemented as an XML file which is parsed and written by Java Architecture for XML Binding (JAXB). A single XML `<Mapping>` object contains, besides an integer-valued attribute for the `priority` of a mapping (the lower the number, the earlier it will be executed), this possible set of elements and attributes (cf. Figure 9.1 for an example):

▷ **Difference** - Name of the change set to map.

- ▷ **Rule** - Rule to map onto difference, contains sub element **Params**.
 name - Name of the rule.
 path - Path to the file in which the rule is contained.

- ▷ **Params** - Container for list of **Param** objects.

- ▷ **Param** - Parameter to set before rule execution, contained in super element **Params**
 name - Name of the parameter to set.

- ▷ **Change** - Sub element of **Param**. Contains the Change of the SemanticChangeSet to extract value of the parameter from.
 name - Name of the Change. One of AddReference, RemoveReference, AddObject or RemoveObject
 type - Optional. Type of the Change. Specifies the changed reference, needed only if name is one of AddReference or RemoveReference.

- ▷ **Reference** - Sub element of the **Change** element. This element contains information about which element to extract value from as most Changes contain one source and one target object.
 name - Name of the Reference. One of src, tgt and obj.
 attribute - Attribute to use from the Reference, e. g., id or name.

```

<Mapping priority="0">
  <Difference>CREATE_State_IN_StateMachine_(top)</Difference>
  <Rule name="CreatedStateInSC" path="platform:/plugin/[...]/SC_DTMC.henshin">
    <Params>
      <Param name="id">
        <Change name="AddObject">
          <Reference name="obj" attribute="id" />
        </Change>
      </Param>
    </Params>
  </Rule>
</Mapping>

```

Listing 9.1. Example rule mapping.

It also exists a form-based Transformation Mapping Editor (see Figure 9.13) for development purposes which shows the recognition rules of all registered SiLift rule bases and the Henshin transformation rules in all projects in the workspace. It can be used to create a basic instance of a transformation mapping XML file. As it is currently not possible to specify parameters in the editor, these files in most occasions have to be edited manually afterwards.

9. Implementation

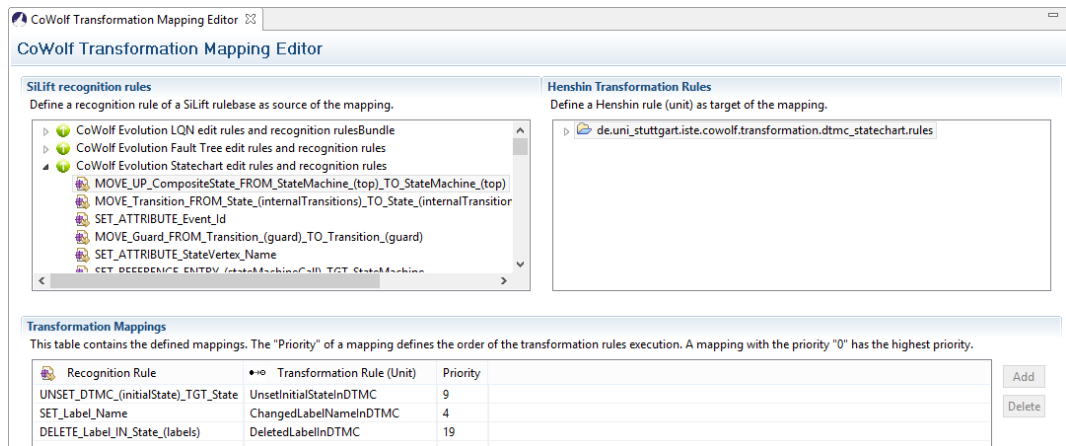


Figure 9.13. Transformation Mapping Editor.

9.5.5 Execution of Henshin-Rules

Author: Rene Trefft, Michael Müller

Henshin rules get loaded by the `AbstractTransformationManager` before the actual transformation starts. Based on the chosen strategy, the execution order of these rules depends:

▷ Rule Mapping

In case of the Rule Mapping, each evolution rule found by SiLift while building the model difference, is mapped to a specific Henshin rule or unit. The order of execution is specified in the rule mapping, for more details see Section 9.5.4.

▷ Working on SiLift Differences

Units or rules specified by the extension point get executed in an unspecified, random order, as they are added by the `AbstractTransformationManager` (see Section 12.3.2).

The rules or units are singularly executed by the `Henshin UnitApplication` that takes a Henshin `EGraph` object, which contains the graph to transform, and a Henshin Engine, where some execution parameters can be set globally, as parameter. By passing a specific Monitor to the Engine, the detailed output log level of the rule execution can be specified. Additionally it is possible to set parameters before every execution of a certain unit/rule. This happens with the method `setParameter` on the rule object. The real execution of the unit/rule happens, when the `execute` method of the `UnitApplication` object gets called.

9.6 Integration Testing

Author: Christian Karl Bernasko

To proof and validate our CoWolf product we needed to test our product. We used manual integration tests as well as the automated graphical testing tool SWTBot. SWTBot is an open-source Java based testing tool for testing Eclipse based applications. SWTBot provides an API to specify a test case. To verify the functionality SWTBot uses the junit test framework. SWTBot specifies for common operations its own set of assertions. If we define a statement with SWTBot, then the SWTBot runner evaluates the statement and throws an exception if it fails. A statement defined with SWTBot looks like the Listing 9.2 below. The statement verifies that a view with the name specified by the variable `cowolf_view` is present.

Listing 9.2. SWTBOT example

```
bot.viewByTitle(_cowolf_view).setFocus();
SWTBotView coWolfView = bot.activeView();
assertTrue(_cowolf_view.equals(coWolfView.getTitle()));
```

SWTBot provides several predefined types which can be used to verify the product. SWTBot reflects the structure of a Eclipse element as a tree. With the SWTBOTTree object it is possible to get elements from a tree, we used this for example to navigate through a tree view of a model or the project structure.

We focused to use the automated integration tests before we made a new release. This verified that the main functionality of the CoWolf project was available and functional in the new release. Since writing an automated integration test need more time then manual tests, we used also manual tests for more complex operations. For a detailed description of the integration tests see Chapter 10.

Acceptance Criteria

An important concept of Scrum is the definition of “Done”. It allows to the stakeholders to assess when the work is complete [SS13]. In order to formalize it, it is necessary to create, within every requirement, acceptance criteria, which are artifacts intended to describe, in a structured way, which are the conditions that the system has to have in order to assert that the requirement is complete.

This chapter shows the acceptance criteria for the automatic and the manual tests. It also shows which criteria was tested by what test.

10.1 Format

Author: Manuel Borja

10.1.1 The Gherkin Description Language

Author: Manuel Borja

Introduction

The description language Gherkin [Cuc13] is used in Behavior Driven Development (BDD) in order to formulate testable requirements. The main advantage of Gherkin is that it is understandable both for stakeholders and developers. Its structure avoids that the requirements contain redundant expressions by establishing pre-defined keywords, namely *given that, and, when, then* and *but*.

Formulation and Syntax

The Gherkin’s structure is established through a set of keywords. These are:

- ▷ Feature: After this keyword a set of properties of a specific functionality will be described. These properties can be written in a unstructured form. Nevertheless a user-story-template style can be used.
- ▷ Background: This keyword allows to describe a specific to-be-done process which is valid for every scenario. Thus, the background has to be performed before the scenario is executed.

10. Acceptance Criteria

- ▷ Scenario: Every feature consists of a set of scenarios which are specified following a common structure. A scenario describes the way the system must work. For that, the system must be put in a specific context, which is defined with the keyword *given that*. How the user has to interact with the system is specified with the keyword *when*. Finally, the expected results follow the keyword *then*.

10.2 Basic Actions

Author: Christian Karl Bernasko

Acceptance criteria ID: AC4.2.1.1
Requirement ID: 4.2.1
Title: Import CoWolf project
Acceptance criteria: Given the user has an existing CoWolf project, which was previously exported And the user has opened the import dialog When the user selects the CoWolf project menu entry And selects the directory path And presses finish. Then the CoWolf project must be imported into the project workspace
Test Case Operations
File → Import → Existing Projects into Workspace

Acceptance criteria ID: AC4.2.1.2
Requirement ID: 4.2.1
Title: Export CoWolf Project
Acceptance criteria: Given the user has an existing CoWolf project in the eclipse workspace And the user has opened the export dialog When the user selects the CoWolf project Menu entry And selects the project And presses finish. Then the CoWolf project should be imported into the project workspace
Test Case Operations
File → Export → CoWolf Project

Acceptance criteria ID: AC4.2.2.3
Requirement ID: 4.2.2
Title: Switching perspective
Acceptance criteria: Given the user is on the Java perspective And selects the Cowolf perspective within the "Open Perspective" dialog. When he selects the CoWolf perspective and clicks OK. Then Eclipse changes into the CoWolf perspective.
Test Case Operations
Window → Open Perspective → Select CoWolf perspective → OK

10. Acceptance Criteria

10.3 Model Editor

Author: Christian Karl Bernasko

10.3.1 TreeView Editor

Author: Christian Karl Bernasko

Acceptance criteria ID: AC4.3.1.0
Requirement ID: 4.3.1
Title: Edit a model in a textual editor
Acceptance criteria: Given the user has created on of the following models (Statemachine, Component, DTMC, CTMC, Faulttree Sequence) And selects one of these models When he opens the context menu "Model Name Model Editor" Then the Treeview editor must be open And the model must be editable within the Treeview editor.
Test Case Operations
Select model → Select contextmenu Model Name Model Editor

Acceptance criteria ID: AC4.5.1.1
Requirement ID: 4.5.1
Title: Edit a Statemachine model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the Statemachine model When the user creates a Statemachine with one or more elements (Statemachine, State, Transition, Composite State, Guard, Event, Transition Action) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.1.3
Requirement ID: 4.5.1
Title: Edit a Component model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the Component model When the user creates a Component Model with one or more elements (Hardware Component, Software Component, Electronic Device, Mechanical Device, Actuator, Sensor, Connector, Prort Instance, Component Instance, Port type, InPort, OutPort) Then the user should be able to create a valid model
Test Case Operations

10.3. Model Editor

Acceptance criteria ID: AC4.5.1.4
Requirement ID: 4.5.1
Title: Edit a CTMC model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the CTMC model When the user creates a CTMC Model with one or more elements (State, Transition, Label) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.1.5
Requirement ID: 4.5.1
Title: Edit a Faulttree model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the model When the user creates a Faulttree Model with one or more elements (Hazard, Or, And, Xor, Prior And, Inhibit, Event Hazard, Event Intermediate Event, Basic Event, Undeveloped Event, Failure Instance, Failure Type, Error Instance, Error Type) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.1.6
Requirement ID: 4.5.1
Title: Edit a Sequence model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the Sequence model When the user creates a Sequence model with one or more elements Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.1.7
Requirement ID: 4.5.1
Title: Edit a LQN model in a Treeview editor
Acceptance criteria: Given the user has opened the Treeview editor for the LQN model When the user creates a LQN model with one or more elements (Model type, Core Type, Processor Type, Slot Type, Run Control Type, Slot Control Type, Solver Params Type, Slot Type) Then the user should be able to create a valid model
Test Case Operations

10. Acceptance Criteria

10.3.2 Graphical Editor

Author: Christian Karl Bernasko

Acceptance criteria ID: AC4.5.2.0
Requirement ID: 4.5.2
Title: Edit a model in a graphical editor
Acceptance criteria: Given the user has created one of the following models (Statemachine, Component, DTMC, CTMC, Faulttree, Sequence) And selects one of these models And opens the context menu "Graphical Editor" When the user creates a new element in the graphical editor Then the user can edit the the element textually in the properties view
Test Case Operations
Select model → Select contextmenu Model Name Model Editor

Acceptance criteria ID: AC4.5.2.1
Requirement ID: 4.5.2
Title: Edit a Statemachine model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the Statemachine model When the user creates a Statemachine with one or more elements (Statemachine, State, Transition, Composite State, Guard, Event, Transition Action) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.2.3
Requirement ID: 4.5.2
Title: Edit a Component model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the Component model When the user creates a Component model with one or more elements (Hardware Component, Software Component, Electronic Device, Mechanical Device, Actuator, Sensor, Connector, Prort Instance, Component Instance, Port type, InPort, OutPort) Then the user should be able to create a valid model
Test Case Operations

10.3. Model Editor

Acceptance criteria ID: AC4.5.2.4
Requirement ID: 4.5.2
Title: Edit a CTMC model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the CTMC model When the user creates a CTMC model with one or more elements (State, Transition, Label) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.2.5
Requirement ID: 4.5.2
Title: Edit a Faulttree model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the model When the user creates a Faulttree model with one or more elements (Hazard, Or, And, Xor, Prior And, Inhibit, Event Hazard, Event Intermediate Event, Basic Event, Undeveloped Event, Failure Instance, Failure Type, Error Instance, Error Type) Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.2.6
Requirement ID: 4.5.2
Title: Edit a Sequence model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the Sequence model When the user creates a Sequence model with one or more elements Then the user should be able to create a valid model
Test Case Operations

Acceptance criteria ID: AC4.5.2.7
Requirement ID: 4.5.2
Title: Edit a LQN model in a graphical editor
Acceptance criteria: Given the user has opened the graphical editor for the LQN model When the user creates a LQN model with one or more elements (Model type, Core Type, Processor Type, Slot Type, Run Control Type, Slot Control Type, Solver Params Type, Slot Type) Then the user should be able to create a valid model
Test Case Operations

10. Acceptance Criteria

10.3.3 (Co-) Evolution

Author: Christian Karl Bernasko

Acceptance criteria ID: AC4.7.1.0
Requirement ID: 4.7.1
Title: Co-evolve context menu
Acceptance criteria: Given the user has created an initial model (DTMC, CTMC, Faulttree, LQN) And the user has evolved the model And the user creates a different model When the user selects the CoWolf context menu "Co-evolve" Then the the two models co-evolve.
Test Case Operations

Acceptance criteria ID: AC4.7.1.1
Requirement ID: 4.7.1
Title: Co-evolve Statemachine model to a Dtmc model manually
Acceptance criteria: Given the user has create a valid Statemachine model And the user has created a Dtmc model When the user selects the CoWolf context menu "Co-evolve" Then the two models co-evolve
Test Case Operations

Acceptance criteria ID: AC4.7.1.2
Requirement ID: 4.7.1
Title: Co-evolve Component model to Faulttree model manually
Acceptance criteria: Given the user has create a valid Component model And the user has created a Faulttree model When the user selects the CoWolf context menu "Co-evolve" Then the two models co-evolve
Test Case Operations

10.3. Model Editor

Acceptance criteria ID: AC4.7.1.3
Requirement ID: 4.7.1
Title: Co-evolve DTMC model to CTMC model manually
Acceptance criteria: Given the user has create a valid DTMC model And the user has created a CTMC model When the user selects the CoWolf context menu "Co-evolve" Then the two models co-evolve
Test Case Operations

Acceptance criteria ID: AC4.7.1.4
Requirement ID: 4.7.1
Title: Co-evolve CTMC model to Faulttree model manually
Acceptance criteria: Given the user has create a valid CTMC model And the user has created a Faulttree model When the user selects the CoWolf context menu "Co-evolve" Then the two models co-evolve
Test Case Operations

Acceptance criteria ID: AC4.7.1.5
Requirement ID: 4.7.1
Title: Co-evolve Sequence model to LQN model manually
Acceptance criteria: Given the user has create a valid Sequence model And the user has created a LQN model When the user selects the CoWolf context menu "Co-evolve" Then the two models co-evolve
Test Case Operations

Acceptance criteria ID: AC4.7.1.6
Requirement ID: 4.7.1
Title: View the difference of two different model types
Acceptance criteria: Given the user has create a valid model And the user has created a model of a different type When the user selects the CoWolf context menu "Version → Show differences" Then the user can review the difference.
Test Case Operations

10. Acceptance Criteria

10.3.4 Analyze

Author: Christian Karl Bernasko

Acceptance criteria ID: AC4.4.0
Requirement ID: 4.4
Title: Analyze context menu
Acceptance criteria: Given the user has created a valid model of the following types (DTMC, CTMC, Faulttree, LQN) When the user selects the CoWolf context menu And selects the analyze menu Then the specific Analyze view opens
Test Case Operations

Acceptance criteria ID: AC4.4.1.1
Requirement ID: 4.4.1
Title: Reliability certification of DTMC or CTMC model
Acceptance criteria: Given the user has created a valid DTMC or CTMC model When the user selects the CoWolf context menu And selects at the Analyze view the verification option And clicks on the finish button Then the verification of the specific model starts
Test Case Operations

Acceptance criteria ID: AC4.4.2.0
Requirement ID: 4.4.2
Title: Performance certification of LQN models
Acceptance criteria: Given the user has created a valid LQN model When the user selects the CoWolf context menu And selects at the Analyze view the solve LQN option And clicks on the finish button Then the performance analyse for the specific LQN model starts
Test Case Operations

10.3. Model Editor

Acceptance criteria ID: AC4.4.3.0
Requirement ID: 4.4.3
Title: Safety certification of Faulttree models with calculation of the top event probability
Acceptance criteria: Given the user has created a valid Faulttree model When the user selects the CoWolf context menu And selects at the Analyze view the "probability of the top event" option And clicks on the finish button Then the probability calculation of the top event starts
Test Case Operations

Acceptance criteria ID: AC4.4.3.1
Requirement ID: 4.4.3
Title: Safety certification of Faulttree models with minimal cutset calculation
Acceptance criteria: Given the user has created a valid Faulttree model When the user selects the CoWolf context menu And selects at the Analyze view the "minimal cutset" option And clicks on the finish button Then the minimal cutset calculation of the Faulttree model starts
Test Case Operations

10. Acceptance Criteria

10.4 Test-Case Description and Execution

Author: Christian Karl Bernasko

In this section we describe the automatic executable test cases that we used to validate the CoWolf project.

T001 Co-Evolution Test

In this test case we used the a statemachine model with the name 00.statemachine. We transformed it initially to a DTMC file. In the next step we evolved the statemachine model to the 01.statemachine model. We Co-Evolved the version 01.statemachine with the DTMC model. Afterwords we verified that the Co-Evolved model equals a recorded verified model.

T002 Project Test

This test asserts that all project items of the CoWolf product are available. The project items are all models, wizards and the creation of the project itself.

T003 Faulttree Analyze Test

This test executes The Faulttree analyzation, it sets the xFTA tool path and asserts that the external tool returns the result.

T004 Editor Test

This test asserts that for every model the corresponding elements can be created and that it is possible to create a valid model.

T005 DTMC Analyze Test

This test executes The DTMC analyzation, it sets the Prism tool path and asserts the the external tool returns the result.

Test Matrix

Testing the product is a critical task. We needed to make sure that all implemented features are functional before we released a new version of CoWolf, we used a test matrix (see Table 10.1) to verify the functionality of the product.

10.4. Test-Case Description and Execution

Table 10.1. Testcase matrix

ID	<i>Manual Testing T001</i>	<i>T002</i>	<i>T003</i>	<i>T004</i>	<i>T005</i>	<i>Success</i>	<i>Failed</i>
AC4.2.1.1	●	-	●	-	-	○	○
AC4.2.1.2	●	-	●	-	-	○	○
AC4.2.1.3	●	●	-	-	-	○	○
AC4.5.1.0	●	-	-	-	●	○	○
AC4.5.1.1	●	-	-	-	●	○	○
AC4.5.1.2	●	-	-	-	●	○	○
AC4.5.1.3	●	-	-	-	●	○	○
AC4.5.1.4	●	-	-	-	●	○	○
AC4.5.1.5	●	-	-	-	●	○	○
AC4.5.1.6	●	-	-	-	●	○	○
AC4.5.1.7	●	-	-	-	●	○	○
AC4.5.2.0	●	-	-	-	●	○	○
AC4.5.2.1	●	-	-	-	●	○	○
AC4.5.2.2	●	-	-	-	●	○	○
AC4.5.2.3	●	-	-	-	●	○	○
AC4.5.2.4	●	-	-	-	●	○	○
AC4.5.2.5	●	-	-	-	●	○	○
AC4.5.2.6	●	-	-	-	●	○	○
AC4.5.2.7	●	-	-	-	●	○	○
AC4.7.1.0	●	-	-	-	-	○	○
AC4.7.1.1	●	●	-	-	-	○	○
AC4.7.1.2	●	-	-	-	-	○	○
AC4.7.1.3	●	-	-	-	-	○	○
AC4.7.1.4	●	-	-	-	-	○	○
AC4.7.1.5	●	-	-	-	-	○	○
AC4.7.1.6	●	-	-	-	-	○	○
AC4.4.0	●	-	-	-	-	○	○
AC4.4.2.0	●	-	-	-	●	○	○
AC4.4.2.1	●	-	-	-	-	○	○
AC4.4.2.2	●	-	-	-	-	○	○
AC4.4.3.0	●	-	-	●	-	○	○
AC4.4.3.1	●	-	-	-	-	○	○

Acceptance criteria ID

Legend

- Test covers criterion
- Test does not cover criterion
- Test result

Evaluation

Author: Michael Müller

This chapter contains the evaluation of the co-evolution component of Co Wolf. Therefore it is compared with a traditional complete transformation of a model in terms of transformation performance and manual work needed to be done after the transformation completed to gain a valid model.

11.1 Goals

In this section the evaluation goals are described in Tables 11.1 and 11.2. The evaluation is done by comparing the incremental transformation between model instances done in the co-evolution with a full transformation for every instance. The transformation used for evaluation is limited to the Statechart to DTMC transformation earlier described in this report.

Table 11.1. Goal 1: Performance of co-evolution

Goal	G1	Determine the performance of the incremental transformation performed by the co-evolution.
	Purpose	Measurement of the performance to allow comparison with full transformation.
	Issue	Performance
	Object	PPU Case Study State Chart model instances are used.
	Viewpoint	User
Question	Q1.1	How well does the incremental transformation process perform in comparison to the complete transformation of a model
	Q1.2	How much is the transformation process influenced by the size of the model?
	Q1.3	Do Henshin rules impact performance analysis?
Metrics	M1	Size of models (Number of nodes and edges) ?
	M2	Execution time in ms
	M3	Number of Henshin rules executed
	M4	Avg. execution time of most expensive Henshin rule (in ms)

11. Evaluation

Table 11.2. Goal 2: Usability of co-evolution

Goal	G2	Determine the effort needed after a full/incremental transformation to create a valid model instance.
	Purpose	To use the model created/modified by transformation, in most cases one has to modify some properties.
	Issue	Usability
	Object	PPU Case Study models previously used to determine the performance of the transformation.
	Viewpoint	User
Question	Q2.1	How many validation errors occur after incremental transformation in comparison to a full transformation?
Metrics	M1	Number of validation errors

11.2 Design

The evaluation is performed as described in this section.

Basic Setup

The basic setup used for the evaluation is given as follows:

- ▷ **Usage of “Pick and Place Unit” Case Study State Chart model instances.** [LFVH13]
This model instances are an example for factory automation systems and are an interesting case for model evolution. For a detailed explanation, see [LFVH13].
- ▷ **Addition of first model instance which is completely empty.**
There was a first model instance added to the series of model instances. This was done to be able to build traces which are in later steps of the co-evolution needed to perform correctly.

Unit test setup

To measure performance/usability of the transformation on this models, the evaluation is executed as two separate unit tests:

1. Perform full transformation on each model instance

For each model instance of the “PPU Case Study” the difference to an empty model is built. Based on this difference, the full transformation to the DTMC model is performed. The evolution step is done with the result that the transformation rules from CoWolf, which are based on the difference, still work. To account for the fact that a full transformation transforms the whole model and does not need the difference, the time interval SiLift takes to build the difference is not included in the measured execution time.

2. Perform co-evolution for each model instance iteratively

Like for the full transformation, first the difference is built. To perform an incremental

transformation now, the difference is essential for the transformation process. Therefore the time SiLift takes to build the difference is part of the whole transformation execution time. Additionally the difference is each time built based on the previous transformation result, while for the full transformation the difference to an empty model is built.

Hardware Setup

The tests were executed on a laptop equipped with

- ▷ Intel Core-i5-460M processor (2 x 2.53 GHz)
- ▷ 4 GB RAM
- ▷ 256 GB SSD
- ▷ no other processes actively running
- ▷ Kubuntu 14.04, 64-bit version

11.3 Threats to Validity

In this section some threats to the validity of the results presented in the next section are taken into account.

▷ SiLift execution times vary much

The results of the co-evolution are influenced by the evolution performed by SiLift. As one can see in the results section, times of evolution execution varies between different instances vary much. Multiple runs however showed that the evolution execution time for each model instance is roughly the same.

▷ Henshin rules impact performance of co-evolution

As one can see in the results section (see Section 11.4), some rules take much more time to be executed than others. While the full transformation only consists of create/delete Henshin rules, which in fact are executed rather fast, the co-evolution also uses some rules which are changing references to objects and get executed rather slowly in comparison. This could be a potential point of improvement for the co-evolution.

11.4 Results

11.4.1 Results for Goal 1

In Table 11.3 and Figs 11.1 and 11.2 one can see the measured values for the evaluation. Please note that the column “Evolution” shows the time needed by the incremental co-evolution to build the difference. It is already included in the total execution time of the incremental transformation used by co-evolution.

11. Evaluation

Table 11.3. Measurements for the performance of the transformation process (M1/M2/M3)

Step	Measured execution times			Executed Rule Count	
	Complete	Incremental	Evolution	Complete	Incremental
1	8,258 s	11,330 s	5,713 s	388	388
2	7,341 s	0 s	0 s	388	0
3	6,680 s	9,424 s	2,987 s	386	16
4	12,052 s	21,638 s	5,892 s	540	326
5	16,323 s	22,138 s	7,138 s	626	292
6	14,986 s	9,597 s	1,102 s	594	58
7	17,444 s	7,242 s	0,949 s	604	46
8	20,980 s	7,300 s	1,041 s	646	72
9	33,576 s	46,299 s	2,129 s	710	148
10	38,825 s	9,589 s	1,455 s	736	40
11	46,052 s	11,200 s	1,628 s	770	118
12	53,897 s	16,244 s	1,784 s	836	104

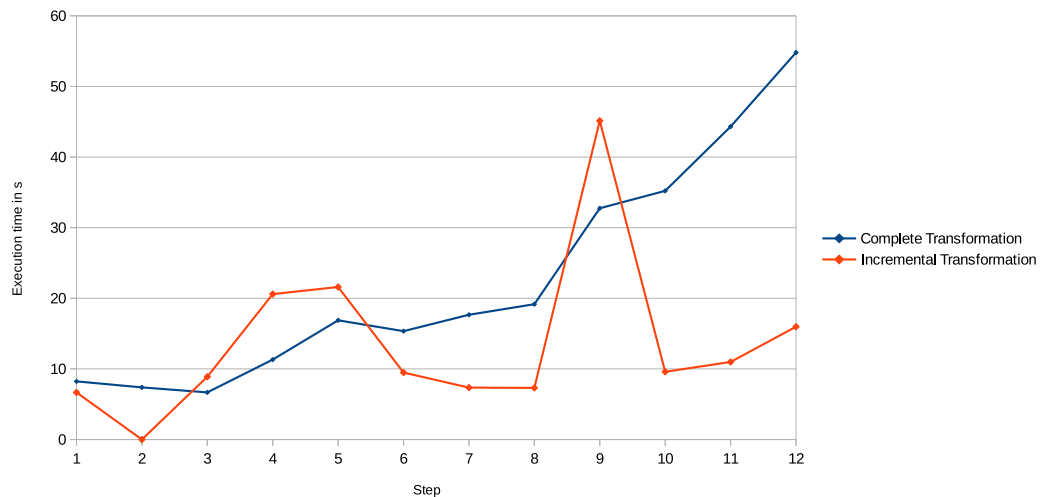


Figure 11.1. Time needed for execution in each transformation step.

As a result it is clearly visible that the incremental process of co-evolution outperforms the full transformation of a single model, as soon as there are relatively little changes in comparison to the model size. In cases where the model is rather small a full transformation can be faster than the incremental co-evolution because in the case of a full transformation no difference has to be built before performing the transformation. Also notable in Figure 11.1 is the outlier in step 9 for the incremental transformation.

11.4. Results

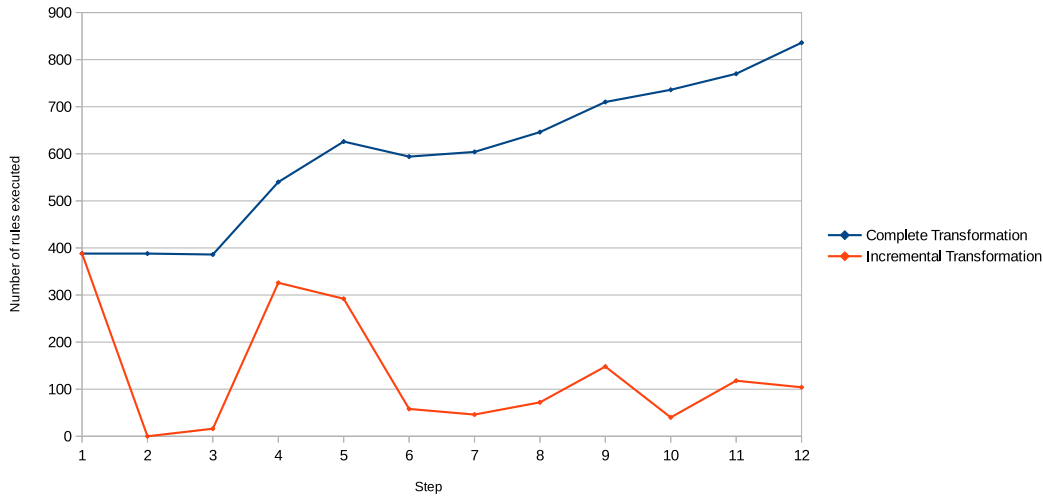


Figure 11.2. Number of rules executed in each transformation step.

While there are in relation to other steps many rules executed, the execution time is still abnormally high. Therefore the evaluation also determines which rule took the longest time to be executed and how often it was executed in total.

Table 11.4. Most costly Henshin rule with total execution time and number of executions

	Complete		Incremental	
1	CreatedTransitionInSC	2,250 s 44	CreatedTransitionInSC	2,412s 44
2	CreatedTransitionInSC	1,446 s 44	-	0 s 0
3	CreatedTransitionInSC	1,690 s 44	ChangedTransitionSource	0,316 s 1
4	CreatedTransitionInSC	5,324 s 62	CreatedActionInSC	4,657 s 22
5	CreatedTransitionInSC	7,602 s 67	CreatedActionInSC	6,909 s 18
6	CreatedTransitionInSC	7,065 s 67	ChangedTransitionTarget	1,493 s 2
7	CreatedTransitionInSC	8,737 s 70	CreatedActionInSC	0,834 s 2
8	CreatedTransitionInSC	11,203 s 77	CreatedActionInSC	1,411 s 3
9	CreatedTransitionInSC	20,626 s 90	ChangedTransitionTarget	20,438 s 15
10	CreatedTransitionInSC	23,401 s 90	CreatedActionInSC	3,849 s 5
11	CreatedTransitionInSC	30,431 s 101	ChangedTransitionTarget	1,822 s 12
12	CreatedTransitionInSC	30,463 s 101	CreatedActionInSC	10,355 s 11

As one can see in the Table 11.4, the most time consuming rule *CreatedTransitionInSC* for the full transformation process always stays the same, while the total time executing it increases relative to the number of its invocations. In contrast to this, the rule which is

11. Evaluation

most time consuming varies for the incremental transformation from step to step. Here often change operations where references are changed occur, which are not utilized by the full transformation of a model. Especially in the already mentioned step 9 the rule *ChangedTransitionTarget* is unusually often called and nearly reaches the execution time of the most time consuming rule for the full transformation. This also explains the outlier in the measurement of the complete transformation process.

11.4.2 Results for Goal 2

Table 11.5. Measurements for the manual amount of work needed after transformation

Step	Number of manual changes needed	
	Complete	Incremental
1	65	65
2	65	0
3	64	1
4	93	41
5	103	30
6	100	4
7	103	5
8	111	10
9	125	16
10	129	5
11	141	14
12	148	11

The results in Table 11.5 and Figure 11.3 clearly state a big advantage for the incremental transformation in the co-evolution process over a full transformation in each step: The number of needed manual adjustments is limited in terms of the co-evolution (for Statechart to DTMC) to cases where a change (create, delete, move) to a transition occurred, while changes made to previous model versions are preserved. In contrast to the incremental process, where probabilities are preserved, the user has to enter probabilities for each transition after a full transformation every time.

11.5 Conclusion

11.5.1 Conclusion for Goal 1

As the computation of the evolution step done by SiLift can be rather time-consuming, for small models it may be practical to use a full transformation of the model, if there's not much manual work to do after the transformation is finished. However, the bigger the

11.5. Conclusion

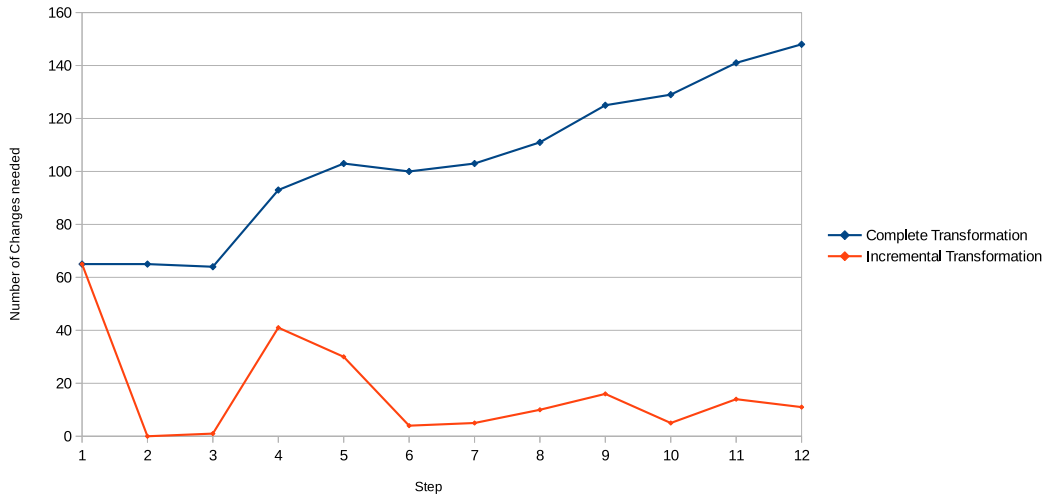


Figure 11.3. Manual adjustments needed after each transformation step

model is, the better the co-evolution scales in comparison with a full transformation of a model instance: While the execution time for the full transformation grows with the model size, the co-evolution execution time grows in general with the number of changes to the original model. The size of the model has only small impact on the execution time for the incremental transformation.

11.5.2 Conclusion for Goal 2

In addition, the incremental co-evolution process preserves manual changes to the target model of the co-evolution, e.g. transition probabilities in a DTMC do not get overwritten by a co-evolution from a state chart model instance. In comparison to a full transformation from a model instance this can save a huge amount of work. While the number of changes of an incremental co-evolution step is in general much lower in comparison with a full transformation executed on the same model, the execution time of a single Henshin rule while performing this step can be much higher. This is the case e. g. when edges change their source or target node.

Developer Guide

In the previous chapters was described how CoWolf is build internally. This chapter now describes how to extend it. It is possible to write your own plug-ins for individual models of your choice. Each model needs plug-ins for its meta model, some for the evaluation and some for the transformation. How to do this is explained in the following sections.

12.1 Develop a New Model

Author: Jonas Scheurich

CoWolf can be adapted with further model types and self-developed metamodels. This section describes how to create a own metamodels suitable for CoWolf.

12.1.1 Create Projects

First, some project has to be created. The conventions for the plug-in's IDs are listed as follows:

- ▷ *[companyID].[modelname]* This plugin project contains the model manager and the generated Java code for the metamodel.
- ▷ *[companyID].[modelname].tests* This fragment project contains the unit tests.
- ▷ *[companyID].[modelname].feature* This feature project bundles all necessary plugins for your model. A feature requirement for supplying it with CoWolf.

The following projects will be generated later:

- ▷ *[companyID].[modelname].emf.edit* This project contains the generated edit operations.
- ▷ *[companyID].[modelname].emf.editor* This project contains the generated EMF tree view editor.

For your own editors:

- ▷ *[companyID].[modelname].[editortype].editor*

12. Developer Guide

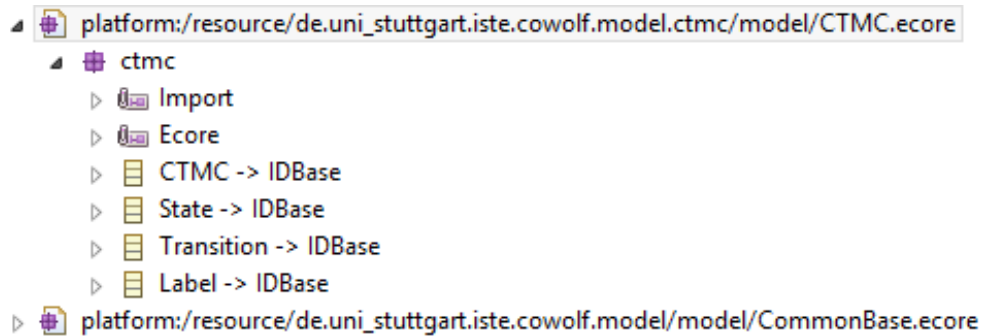


Figure 12.1. Ecore classes are inherit from *IDBase* in the *CommonBase* meta model to create identifier.

12.1.2 Create Ecore Model

Now create your metamodel in a "model" folder (convention) in the `[companyID].[modelName]` project. Add the meta model file to the build properties. Further information about meta model development can found on the EMF project page [Emf]. Ensure that your meta model contains a root class and be aware, that every class must be contained in a another class (except the root class).

To perform the evolution all ecore classes need an identifier. Import the *CommonBase* ecore metamodel into your metamodel and make the ecore classes inherit from *IDBase* and the IDs will be generated automatically for every object instance (see Figure 12.1).

12.1.3 Create Genmodel and Generate Editor

Now create a genmodel file in the "model" folder. This file contains all properties for the generation of the Java code. Import your metamodel (see Figure 12.2). create a

If you use in your ecore metamodel a third party ecore metamodels refer their genmodel (see second table in Figure 12.2).

Now configure the genmodel. In the following list we describe some configurations. For more details see the CoWolf project wiki.

Properties of the Meta Model in the Genmodel

- ▷ **BasePackage:** Change to conventions described in 12.1.1.
- ▷ **File Extensions:** Set your file extension.
- ▷ **Package Suffixes:** Change to conventions described in 12.1.1.

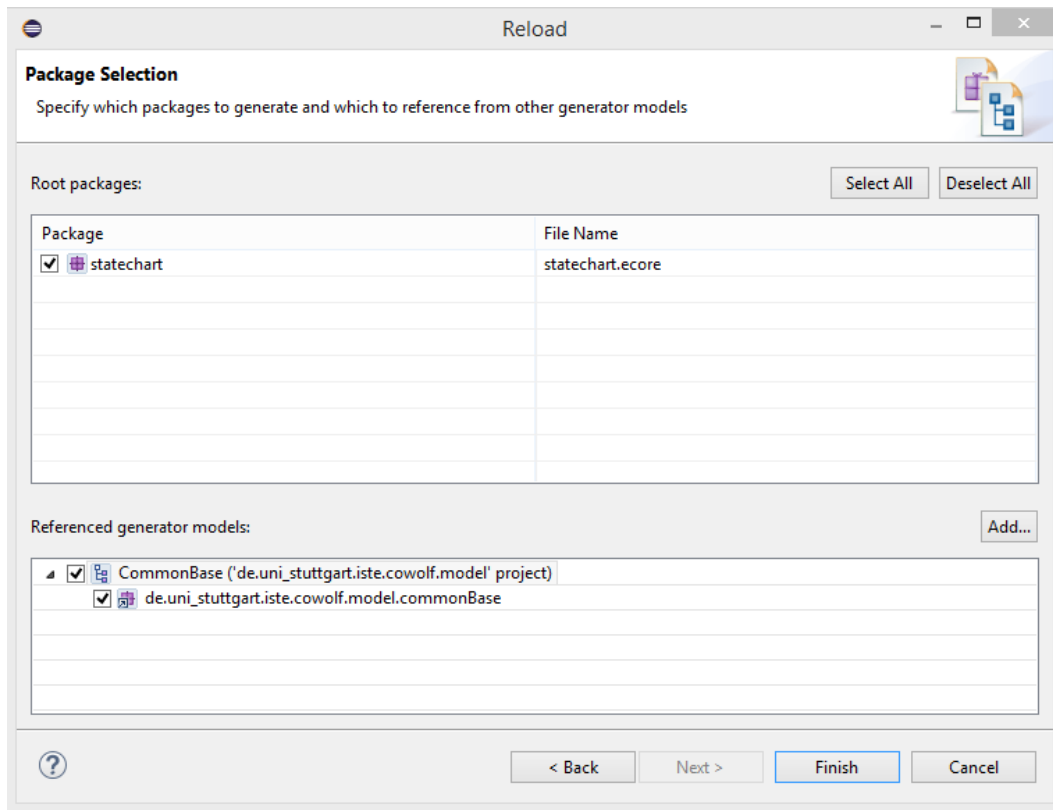


Figure 12.2. Import of a.ecore metamodel into an genmodel.

- ▷ **Generate Example Class:** If you don't want the example classes set this property to *false*;

Properties of the Genmodel

- ▷ **IDs and class names:** Change the class names like the conventions described in 12.1.1.

Now generate the Java code for model, edit and editor. You find the options in the context menu of the root element in the genmodel. If your metamodel inherit from many.ecore classes be aware, that the Java implementation inherits from the first class in the inherit list and the other classes are only implemented by their interfaces, so constructors of the second inherited.ecore class and so on are not called automatic.

12.1.4 Change the Wizard

EMF generates beside the editor an Eclipse wizard for the creation of a new model instance. In the root elements dropdown are all classes available. You have to be sure that only your root element can be selected. Change the following method into the `[modelName]ModelWizard` class like the following example:

Listing 12.1. Code to remove in the model wizard

```
@Override
protected Collection<String> getInitialObjectNames() {
    if (initialObjectNames == null) {
        initialObjectNames = new ArrayList<String>();
        for (EClassifier eClassifier : dtmcPackage.getEClassifiers()) {
            if (eClassifier instanceof EClass) {
                EClass eClass = (EClass)eClassifier;
                if (!eClass.isAbstract() && eClass.getName().
                    equalsIgnoreCase("[TODO_Change_This_to_root-node]")) {
                    initialObjectNames.add(eClass.getName());
                }
            }
        }
        Collections.sort(initialObjectNames, CommonPlugin.INSTANCE.getComparator());
    }
    return initialObjectNames;
}
```

Ensure that the *generated* annotation is changed to *generated NOT* to prevent your changes when the Java code is regenerated.

12.1.5 Implement *AbstractModelManager*

To connect your model with CoWolf, implement the `AbstractModelManager` in the `[companyID].[modelName]` project. The abstract methods to be overwritten are listed as follows:

- ▷ **getManagedClass()**: The class type of your root element. e.g. `CTMC.class`;
- ▷ **getModelName()**: The display name of your model.
- ▷ **getModelNamespace()**: The namespace of your model. It can be accessed with the `eNS_URI` attribute of the metamodel package class.
- ▷ **getFileExtension()**: The file extension, as defined in the `genmodel`.

12.2. Develop a New Evolution

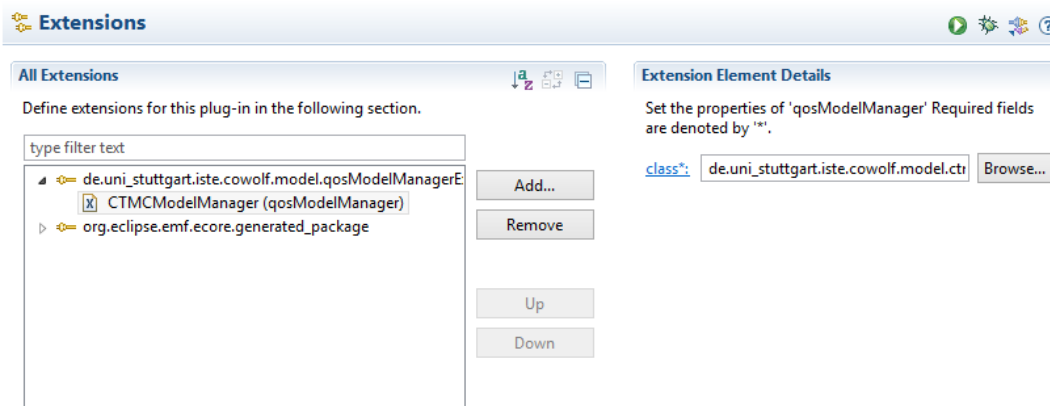


Figure 12.3. Create a *ModelManagerExtension* to provide your *AbstractModelManager* implementation.

Create the Extension Point

The model manager will be provided to Co Wolf by an extension point. Create an *ModelManagerExtension* to provide your *AbstractModelManager* implementation (see Figure 12.3).

12.1.6 Add a Graphical Editor

Author: Verena Käfer

A graphical editor for a model is a nice to have but no must have. If you want to create a graphical editor for your model you need a new plugin project for it. As described in 9.1.2 a graphical editor is defined in an *odesign* file. That section also describes how an *odesign* file is built. What you now want to do is to add the creation of the *aird* file to your model creation wizard. For this, please have a look at the model creation wizards in the editor plugins of the other models. Here you can easily see how the connection can be done.

If your root element for the viewpoint is not the top element in your model, then (and only then) you need to override the `getRootObject()` method in your model manager. This can be seen in the model manager of the Sequence Diagrams.

12.2 Develop a New Evolution

Author: Michael Zimmermann

This chapter will explain how to extend the evolution feature of Co Wolf to be able to show the evolution of another model. As already mentioned in Chapter 8.3, Co Wolf makes use of SiDiff and SiLift to realize the evolution feature. Hence, most of the requirements to develop the evolution feature for a new model are SiDiff- and SiLift-specific. Anyway, all necessary SiDiff/SiLift as well as the Co Wolf-specific steps will be explained here.

12. Developer Guide

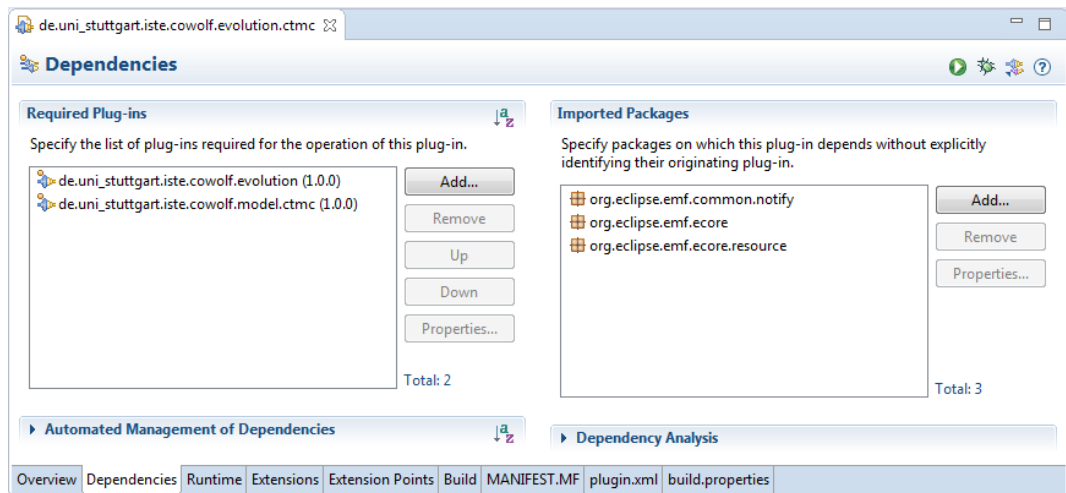


Figure 12.4. MANIFEST.MF → Dependencies for EvolutionManager.

Important: This guide is based on SiLift version 0.1.0.201409111244. In another version of SiLift the required steps could be different as shown here.

1 Creation of the Evolution Manager

Every model needs its own EvolutionManager. The manager is needed to enable the evolution feature for the specific model. In addition, the matcher that should be used to calculate the correspondences for this model is specified here. The choice of the matcher depends on the model. If, for example, the model elements have UUID attributes, the ID-based matcher should be used (see chap. 5.1.5). Before creating the EvolutionManager, a new Plug-in Project has to be created first. For this, go to File → New → Other... → Plug-in Development and create a Plug-in Project. Then, open the MANIFEST.MF, go to Dependencies and add the dependencies shown in Figure 12.4.

Important: This example is taken from the CTMC model. Instead of `de.uni_stuttgart.iste.cowolf.model.ctmc` the desired model plug-in must be added.

After that, the EvolutionManager must be implemented. Listing 12.2 shows the implementation of the EvolutionManager for the CTMC model example. Important lines are commented.

Afterwards this plug-in needs to be defined as a CoWolf extension. Therefore, open the MANIFEST.MF again and go to Extensions. Click on Add... and select the `de.uni_stuttgart.iste.cowolf.evolution.evolutionManagerExtension` extension point there. Click on Finish to add it. Afterwards right click on it, go to New and click on `evolutionManager`. Now, on the right side under Extension Element Details, you have to define your previously implemented EvolutionManager class. This result should look similar like the

```

public class CTMCEvolutionManager extends AbstractEvolutionManager {
    @Override
    public boolean isManaged(Resource model) {
        if (model == null || model.getContents() == null
            || model.getContents().isEmpty()) {
            return false;
        }

        return model.getContents().get(0) instanceof CTMC;    // Your
                                                                model
    }

    @Override
    public EvolutionTypeInfo getEvolutionTypeInfo() {
        EvolutionTypeInfo info = new EvolutionTypeInfo();
        // Specify matcher you want to use
        info.setMatcher(EvolutionTypeInfo.MATCHER_EMFCOMPARE);
        return info;
    }

    @Override
    protected Class<? extends EObject> getManagedClass() {
        //Specify main model class
        return CTMC.class;
    }
}

```

Listing 12.2. Example EvolutionManager implementation for CTMC.

example in Figure 12.5.

2 Creation of the Technical Difference Builder

The next step is to implement the `TechnicalDifferenceBuilder` for this model. This class is needed in order to build the low-level difference. Furthermore, model elements, which should be filtered so they are not part of the resulting low-level difference, can be defined here. Like described in the previous step, first create a new `Plug-in Project`, go to `MANIFEST.MF` → `Dependencies` and add the dependencies shown in Figure 12.6. Again, instead of `de.uni_stuttgart.iste.cowolf.model.ctmc` your desired model must be added.

Next, the `TechnicalDifferenceBuilder` must be implemented. Listing 12.3 shows an example implementation of the `TechnicalDifferenceBuilder` for the CTMC model.

In order SiLift can use the `TechnicalDifferenceBuilder`, this plug-in needs to be registered as an extension. This time add the extension point

12. Developer Guide

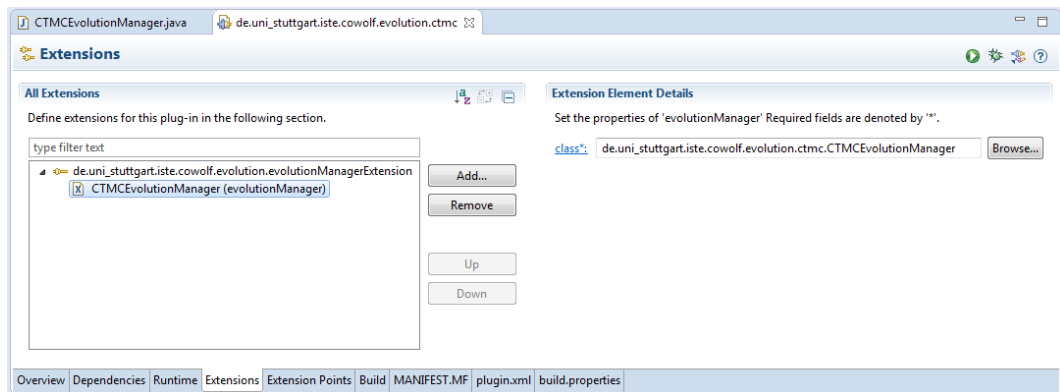


Figure 12.5. MANIFEST.MF → Extension for EvolutionManager.

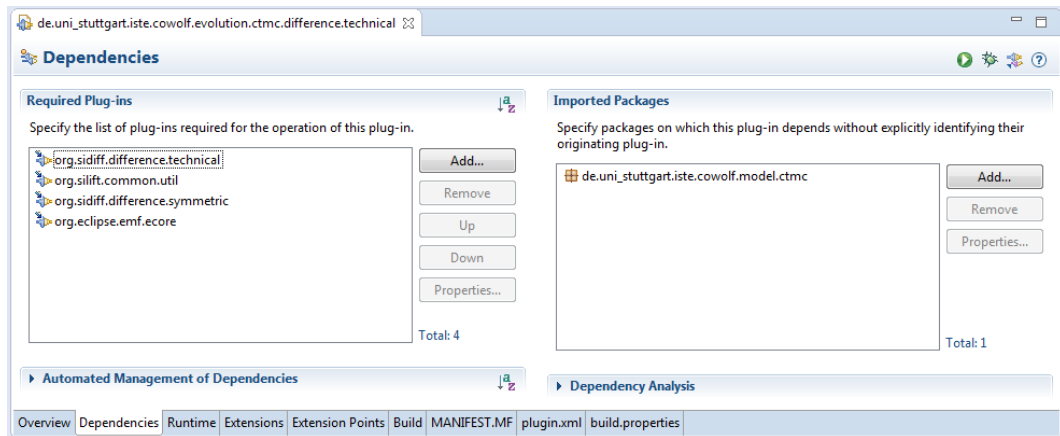


Figure 12.6. MANIFEST.MF → Dependencies for TechnicalDifferenceBuilder.

org.sidiff.difference.technical.technical_difference_builder_extension. Afterwards right click on it, go to New and click on technical. Now again on the right side under Extension Element Details insert your previously implemented TechnicalDifferenceBuilder class.

3 Creation of the RuleBase Plugin-Project

This SiLift plug-in contains all recognition rules for the specific model. They enable the possibility of SiLift to detect user edit operations. The recognition rules are Henshin rules (see chap. 5.1.4) and are automatically generated from SiLift if the associated edit rules are located in the same plug-in.

Go to File → New → Other... → SiLift and click on RuleBase Plugin-Project. Specify

```

public class TechnicalDifferenceBuilderCTMC extends TechnicalDifferenceBuilder {

    @Override
    protected Set<EClass> getUnconsideredNodeTypes() {
        //NodeTypes can be filtered here
        return null;
    }

    @Override
    protected Set<EReference> getUnconsideredEdgeTypes() {
        //EdgeTypes can be filtered here
        return null;
    }

    @Override
    protected Set<EAttribute> getUnconsideredAttributeTypes() {
        //AttributeTypes can be filtered here
        return null;
    }

    @Override
    protected String getObjectname(EObject obj) {
        return obj.toString();
    }

    @Override
    public String getDocumentType() {
        return CtmcPackage.eNS_URI;    // Adapt this to your model
    }

    @Override
    public String getName() {
        return "CTMC_Technical_Difference_Builder";
    }
}

```

Listing 12.3. Example TechnicalDifferenceBuilder implementation.

the name and other settings and click on Next until you come to the Templates selection. Here, choose RuleBase Plugin-Project and click on Finish. This time the required dependencies and the extension should be added to the MANIFEST.MF automatically.

Now, you can place your edit rules for your model into the editrules folder of the

12. Developer Guide

created `RuleBase Plugin-Project`. Atomic edit rules can be generated using the `SiDiff Edit Rule Generator (SERGe)` [Rin14]. To do this, you need a `SERGe` configuration file for your model. It's a XML file with the extension `serge` specifying parameters for the generation. Listing 12.4 shows the configuration file used for the `CTMC` model. To generate the edit rules right click on the configuration file go to `SERGe` and click on `Generate CPEOs`. This will open the wizard to generated the edit rules.

Important: In order to generate the edit rules with `SERGe` your model needs to be installed in your Eclipse instance.

As already mentioned, `SERGe` can only generate atomic edit rules. Needing some complex ones you have to create them manually. If you want to do this, you have to pay attention to some points:

- ▷ The file containing an edit rule has to end with `_execute.henshin`.
- ▷ Every edit rule must be located in a separate file.
- ▷ Every edit rule needs a *mainUnit*.

Now, that the edit rules are located in the `RuleBase Plugin-Project` only the recognition rules are still missing. `SiLift` generates them as well as the `rulebase` file during the build process. To trigger the generation manually go on `Project` and click `Clean...` (`Build Automatically` should be enabled). After the workspace was built, the recognition rules should be located in the `recognitionrules` folder and additionally a `rulebase` file should be located in the root of the plug-in. The `rulebase` file can be opened with the `Rulebase Editor` to manage the recognition rules (e. g., disable or enable them).

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Config SYSTEM "http://pi.informatik.uni-siegen.de/SiDiff/
  Editrulesgeneratorconfig.dtd" >
<Config>
  <GeneralSettings>
    <preventInconsistency value="true" />
    <multiplicityPreconditions integrated="true" separately="false" />
    <reduceToSuperType      ADD_REMOVE="true" CHANGE_LITERAL="true"
      CHANGE_REFERENCE="true" CREATE_DELETE="true" MOVE="true"
      MOVE_DOWN="true" MOVE_UP="true" SET_UNSET_ATTRIBUTE="true"
      SET_UNSET_REFERENCE="true" />
    <disableVariantsWithSupertypeReplacement value="false" />
    <modelUsesProfileMechanism value="false" />
    <createAllAttributes value="true" />
  </GeneralSettings>
  <MetaModelSettings>
    <!-- Specify your model nsURI here-->
    <MainModel nsUri="http://ctmc/1.0" />
    <MaskedClassifiers/>
  </MetaModelSettings>
  <Transformations>
    <Creates allow="true" />
    <Deletes allow="true" />
    <Moves allow="true" allowReferenceSwitching="true"
      allowReferenceCombinations="true" />
    <MoveUps allow="true"/>
    <MoveDowns allow="true"/>
    <SetAttributes allow="true" />
    <UnsetAttributes allow="true" />
    <SetReferences allow="true" />
    <UnsetReferences allow="true" />
    <Adds allow="true" />
    <Removes allow="true" />
    <ChangeLiterals allow="true" allowLiteralSwitching="true" />
    <ChangeReferences allow="true" />
  </Transformations>
    <!-- Specify your model root element-->
  <Root name="CTMC" nested="false"/>
  <WhiteList>
    <!-- You can whitelist elements here-->
  </WhiteList>
  <BlackList>
    <!-- You can blacklist elements here-->
  </BlackList>
</Config>

```

Listing 12.4. Example SERGe configuration file.

12.3 Develop a new Co-Evolution

Author: Philipp Niethammer

The main functionality of CoWolf is the transformation between different model types. The continuous transformation between associated models to follow up changes in one model is then called co-evolution.

As these transformations are very specific depending on source and target models, each transformation must be build on its own. CoWolf provides two easy-to-use methods to define transformations. Additionally, the developer may define a custom method, e.g., to integrate an existing transformation that for instance is not using Henshin at all.

This section describes the transformation framework of CoWolf in general, the usage of the two implemented methods and how to create custom transformation methods.

12.3.1 Transformation Framework

Each transformation is created as a Eclipse feature that may contain one or multiple Eclipse plugins. A transformation feature can provide the transformation between two model types either one- or bidirectional. But at the moment, in a user's installation there must only be one method for each (directed) transformation.

Preconditions and Conventions

To run a transformation, the following preconditions must be met:

- ▷ There must be a Model Plugin implemented and installed for both models
- ▷ For each possible source model, there must be an evolution plugin implemented.
- ▷ Each transformation feature must implement the `AbstractTransformationManager` and implement at least the following 4 abstract methods:
 - ▷ `getManagedClass1` - returns the model root class of one side of the transformation.
 - ▷ `getManagedClass2` - returns the model root class of the second side of the transformation.
 - ▷ `getKey` - returns a (installation unique) string that identifies the transformation from `getManagedClass1` to `getManagedClass2`.
 - ▷ `getReverseKey` - A (installation unique) string that identifies the transformation from `getManagedClass2` to `getManagedClass1`.
- ▷ If the transformation is unidirectional, the manager must also override the method `isManaged(Class<?> source, Class<?> target)` so that it returns true if and only if source and target matches the directional transformation.

12.3. Develop a new Co-Evolution

- ▷ The transformation manager must register itself to the system by adding the extension point `de.uni_stuttgart.iste.cowolf.transformationManagerExtension`.

Additionally, we define the following conventions. These are not necessary for the execution of the transformation, but increase code quality in terms of readability and consistency.

- ▷ The package name of all plugins should contain `transformation.modelA_modelB`, prefixed by your company/product domain, suffixed with the sub packages.

For example: `de.uni_stuttgart.iste.cowolf.transformation.dtmc_ctmc` for the transformation between DTMC and CTMC.

- ▷ The key and reverse key should contain source and target model name.

For example, the key of the DTMC/CTMC transformation is `ctmc_dtmc`, describing the transformation from CTMC to DTMC, and the reverse key is `dtmc_ctmc` analogous. Concerning future improvements (cf. Section 14.2.2), it might be good a practice to include a name of the transformation method or provider to ensure uniqueness in case of multiple plugins for the same transformation.

12.3.2 Defining Transformation Rules

The transformation framework uses the SiLift `SymmetricDifference` provided by an evolution plugin. The transformations are defined with Henshin.

Traces

The traces provided by Henshin are the connection between a model instance of one side of the transformation and the other side of the transformation. A trace contains one or more source- and target-references. By convention, the **source** reference of a trace must be connected to an object contained in the `getManagedClass1` meta model, and the **target** reference of a trace must be connected to an object contained in the `getManagedClass2` meta model. This is irrespective of the direction of the performed transformation. Subtraces are not supported by the transformation framework.

It is worth noting that the traces are automatically treated to avoid side effects in the trace's references when working with model versions or moving the referenced model instances on the file system. In this process, the file part of the reference is replaced by a virtual identifier. This is `transform:source` for the instance of `getManagedClass1` and `transform:target` for the instance of `getManagedClass2`. Information provided by the `ModelAssociationManager` (9.2) is used to resolve these identifiers again. However, this conversation is transparent in regular development.

12. Developer Guide

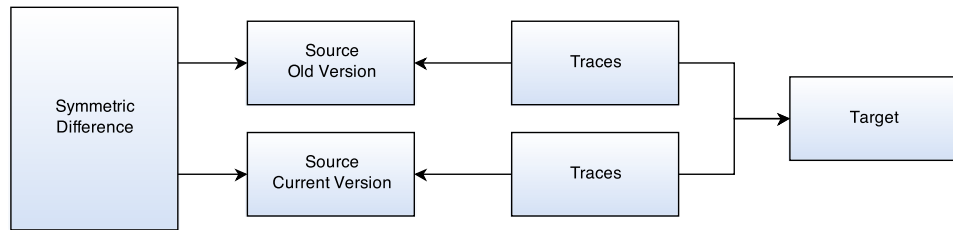


Figure 12.7. Components of the transformation graph

Provided Graph for a Transformation

Henshin works on an EGraph, containing all resources needed in the transformation rules. This graph is automatically generated by the transformation framework and contains the following components.

- ▷ The base version of the source model instance, which is used as point of origin to identify changes.
If there was a transformation between these model instances in either direction before, this version is selected. Otherwise, the initial version of the source instance, containing only the root node of the model, is used.
- ▷ The current version of the source model instance.
- ▷ The current version of the target model.
- ▷ Traces between the base source version and the target, if available from the last co-evolution.
- ▷ Traces between the current source version and the target.
This is a copy of the traces between the base source version and the target that is resolved against the current source version. It also forms the basis for the resulting traces.
- ▷ The `SymmetricDifference`, containing information about what changed between the base and the current source version.

In short, for every detected change there is an object that identifies the type of modification (e.g. `AddObject`, `DeleteObject`, `AddReferenceObject`, etc.). This object references to either an object in the base source version (in case of deletion), the current source version (in case of creation) or both (in case of modification). Additionally, it also contains informations about correspondences in the base and the current source version and about liftings found. For more details, please refer to the sections 5.1.5, 5.1.6 and the SiLift documentation [Sof14b].

A diagram of all components and their interrelation is shown in Figure 12.7.

Method 1: Mapping

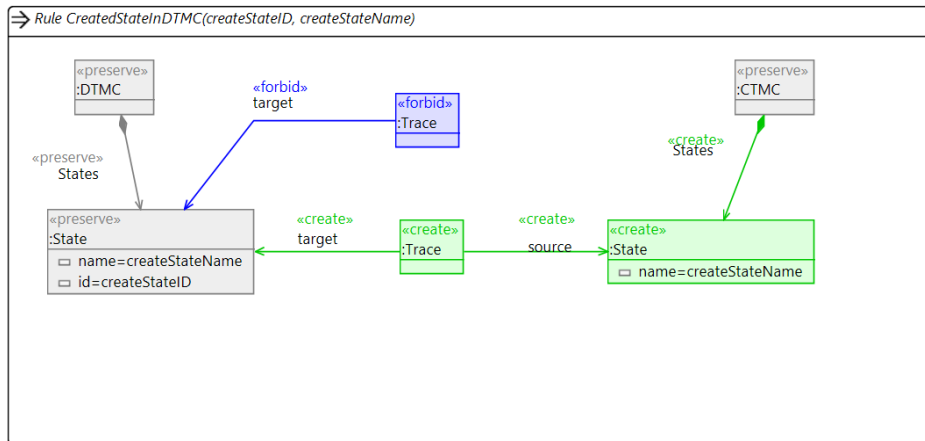


Figure 12.8. Example of a mapping based transformation rule

SiLift detects changes in the source model by so-called recognition rules. This method works by mapping each of recognition rules to a Henshin rule that performs the transformation for this change. This is done by comparing the name of the rule with the name specified in the mapping. For every occurrence in the difference the specified Henshin rule or unit is executed. Before executing the matched rules/units, these are sorted by priority. This is needed because some rules have to be executed before others, otherwise the transformation could produce false results or abort in some cases. If, for example, a transition is created pointing to a new state, the state must be created before the transition is.

In the specification of the mapping, parameters found by the recognition rule can be passed to the transformation rule to identify the changed elements in Henshin. For example, a created state in the source model can be found by its ID in Henshin and attributes like name can be copied to the created state in the target model. Figure 12.8 shows how such a Henshin rule can look like.

We always provide both source models, the old model (before the changes are made) and the new model, thus both, created and deleted elements can be matched by their ID or other properties defined by the mapping.

To use this method, the extension *de.uni_stuttgart.iste.cowolf.transformationMappingExtension* must be added to the plugin for each direction, defining the direction key (either *getKey* or *getReverseKey*) and the mapping file relative to the project root. For details how the mapping can be specified in an external XML file, see Section 9.5.4.

Although this method is fully supported and is used by several of our provided transformations, we do no longer recommend this method but to use the following method

12. Developer Guide

of directly working with the difference set. By using an external mapping file, the context between the detected difference and the performed action is not obvious in the Henshin rule file. Additionally, the names of the recognition rules may change in later development and the mapping needs to be changed to match them again.

Method 2: Working on SiLift Differences

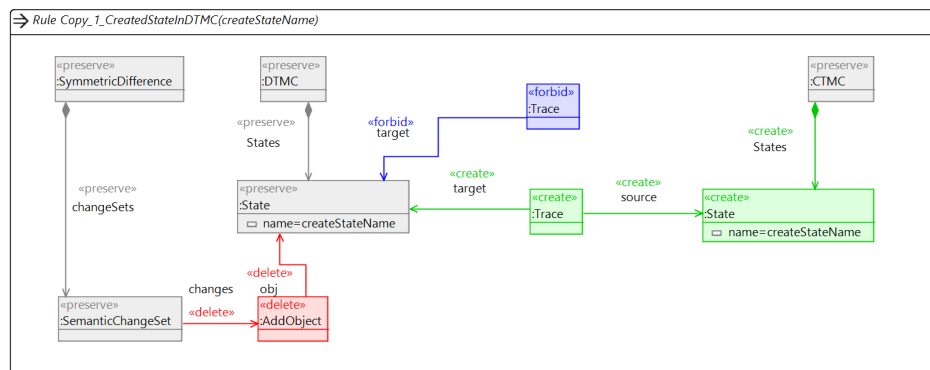


Figure 12.9. Example of a transformation rule using SiLift Differences

The aim of this method is to encapsulate all knowledge about the transformation into the Henshin file. Therefore we provide the difference model from SiLift in the Henshin graph. A Henshin rule now uses a `ChangeObject` from this information to identify the changes in the source model, as shown in Figure 12.9. This allows the usage of *multi-rules* for each operation which often makes the transformation rules smaller and performs better than the rule mapping of the above method.

CoWolf calls only one unit in the Henshin file, which is named `mainUnit` by default. This unit must define the complete course of transformation, calling all rules in the needed order. This method also allows to define more complex execution rules, for example based on conditions or even recursive calls.

To use this method, the extension `de.uni_stuttgart.iste.cowolf.transformationRuleExtension` must be added to the plugin for each direction, defining the direction key (either `getKey` or `getReverseKey`) and one or more Henshin files (relative to project root) with their main method. For very complex transformations, we added the possibility to separate the rules to several Henshin files. These are called in the order of definition in the extension.

Defining a Custom Method

For transformations that can't be done with one of the methods above and for integrating existing transformations, the process can be customized by overriding the method

12.3. Develop a new Co-Evolution

`runTransformation(ResourceSet, SymmetricDifference)` in the transformation manager class.

The EMF resource set contains all resources that are available for transformation, as listed below. It is elementary to work only on these resources to avoid unwanted dependencies and concurrency problems. To access one of the resources, `ResourceSet.getResource(URL, false)` is used. The second parameter of this function defines, if a resource may be loaded automatically on demand. This option as well as any other file system operations (load, save) must not be used on the resources, as the resources are pure virtual and those operations will fail.

The following resources are available. They are best accessed by the given constant URLs or getter methods.

- ▷ `RESOURCE_URL_OLD`: The old source model.
- ▷ `getSourceUri(ResourceSet)`: The current source model.
- ▷ `getTargetUri(ResourceSet)`: The target model.
- ▷ `RESOURCE_URL_OLDTRACES`: The traces leading from the old source model to the target model.
- ▷ `RESOURCE_URL_TRACES`: The traces leading from the current source model to the target model.
- ▷ `RESOURCE_URL_DIFF`: The SiLift differences.
- ▷ `RESOURCE_URL_RESULT`: The resulting target model.

The SiLift differences are additionally given as second parameter to the method.

After performing the transformation, the new target model is to be saved in `RESOURCE_URL_RESULT` and the current traces in `RESULT_URL_TRACES`. These resources are then automatically saved as the result.

For easier usage, the following methods may help:

- ▷ `generateGraph` generates a `EGraph` from the `ResourceSet`, that is needed for Henshin.
Generating the graph with all dependencies is not a trivial task, as explained in Section 9.5.2. We really recommend to use this method instead of creating the graph by oneself. Although, the method only adds the default resources, named above. If custom resources are used, they need to be added into the graph afterwards.
- ▷ `extractResultFromGraph` extracts the resulting target model and traces from the graph and saves them in the resource set.

Usually, if there are multiple transformations to perform, they are executed in sequence to avoid problems with concurrency. Though, there is no warrant for that. Therefore, there must not be used any class variables (fields) for dynamic content in the transformation but all information should be passed as parameter. Another way is, to build an external transformation class and create an instance of this class in `runTransformation`.

Use of Co Wolf

This chapter explains how to use CoWolf. It is thought as instruction manual for the user. It starts with an explanation of the installation process and then shows how to create a new model instance. Later it is described how to export models and how the internal version management works. Finally, the usage of the main functions are shown: The evolution, transformation and analysis of a model.

13.1 Installation

Author: Christian Karl Bernasko

The CoWolf IDE integration requires Eclipse 3.4 (alias Luna) or higher and a Java JRE 7 or higher. In this section we will guide you through the installation process. The CoWolf project comes with three installation options:

- ▷ use the Eclipse product
- ▷ install from the update site
- ▷ compile from source

13.1.1 CoWolf Product

Author: Christian Karl Bernasko

To get a full eclipse together with CoWolf we provide the following URL:

<https://github.com/DevProject552014/CoWolf/releases>

13.1.2 Install from the Update Site

Author: Christian Karl Bernasko

If you already have an Eclipse 4.4 running you should install the CoWolf dependencies (Henshin, SiLift, Xtext) from one of the update sites listed below. To do so within Eclipse choose Help → Install New Software.... In the upcoming dialog you should paste one of the update site URL's into the field named work with. Click on the button "Select All" and click "Next" and on the next page "Finish".

CoWolf **Dependencies:**

13. Use of Co Wolf

- ▷ CoWolf makes use of Henshin in order to execute transformations between models.
<http://www.eclipse.org/henshin/>
- ▷ CoWolf makes use of SiLift in order to calculate the differences between evolved models.
<http://pi.informatik.uni-siegen.de/Projekte/SiLift/>
It is **recommended** to use the latest stable version of SiLift for CoWolf:
<http://cowolf.github.io/SiliftUpdateSite/>
- ▷ CoWolf makes use of Sirius in order to edit the models in a graphical representation.
<http://www.eclipse.org/sirius/index.html>
- ▷ CoWolf makes use of Xtext in order to verify PCTL rules.
<http://www.eclipse.org/xtend/>
- ▷ The last step is to install the Co Wolf product. Use the following URL to install Co Wolf:
https://github.com/DevProjectSS2014/p2_update_site/raw/master

13.1.3 Compiling CoWolf from Source

Author: Christian Karl Bernasko

In this section we will explain how to setup your development environment and how to compile Co Wolf from source. We will first explain the setup process in general and then will be going through a detailed step by step description of how to setup the environment. At first we must install the fundamental tools, which are Git, Maven 3 and Eclipse. Then we import the server certificate into the java keystore. This lets us communicate with the Lismore server. Then we are ready to compile Co Wolf with the Maven standalone version. In the next steps we are configuring the Eclipse IDE to make it able to compile the source code. We will install several plugins and configure the target platform. After this tasks we can compile Co Wolf.

Fundamental Build Tools

Download and install the following required tools:

- ▷ Eclipse 4.4 SDK (alias Luna)
- ▷ Apache Maven 3
- ▷ Git

Import the Server Certificate

We will use maven as our build tool of choice, in order that maven is able to communicate with the Nexus artifact repository. The certificate is required because our server lismore.informatik.uni-stuttgart.de provides only a self-signed certificate. Maven depends

on java and can only communicate with the Lisemore server of University Stuttgart, if the certificate of the Lisemore server is installed in the Java key store. This means you need to accept the certificate before java can communicate with the server. Download the following certificate from our website:

```
http://cowolf.github.io/lismore.informatik.uni-stuttgart.de.crt
```

Once the certificate is downloaded, install it with the following command into your java keystore.

```
keytool -importcert -noprompt -keystore
<path-to-java-jre>/lib/security/cacerts -storepass changeit -alias
lismore.informatik.uni-stuttgart.de -file lismore.informatik.uni-stuttgart.de.crt
```

Listing 13.1. Self signed cert import.

Path to Java JRE:

- ▷ Windows: <JRE-Install-Dir>/bin
- ▷ Linux: /usr/lib/jvm/java-7-openjdk-amd64/jre/

When you finished this task, java is able to communicate with the lismore.informatik.uni-stuttgart.de server.

Compile with Maven Standalone

At this point you are able to compile the CoWolf source code with the maven build tool. Maven will download all dependencies and install them into then .m2 repository within your home directory. To compile the source code with the maven standalone version, clone the repository from Github with the following command:

```
Git clone https://github.com/DevProjectSS2014/CoWolf.git
```

Then change into the directory `de.uni_stuttgart.iste.cowolf.parent`. In this directory type the command `mvn clean verify -X -e`. The command will build the CoWolf source code. The result is a p2 repository which can be installed into an Eclipse IDE. To install the p2 repository use the following directory: `de.uni_stuttgart.iste.cowolf.p2update/target/repository/`

Compile within the Eclipse IDE

Within a running Eclipse IDE you should install the CoWolf dependencies (UML2, SiLift, Xtext, Henshin, EGit, M2E, RCP, EMF, Sirius, Swtbot, Maven Development Tools, OCL) from one of the update sites listed below. To do so within Eclipse choose Help → Install New Software.... In the upcoming dialog you should paste one of the update site URL's into the field named Work with. Click on the button Select All and click Next and on the next page Finish.

- ▷ UML2: Luna Update-Site: UML2 Extender SDK

13. Use of Co Wolf

- ▷ SiLift: <http://cowolf.github.io/SiliftUpdateSite/>
- ▷ Xtext 2.6.2 (also installs Xtend): Eclipse Marketplace <http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases>
- ▷ Henshin: <http://download.eclipse.org/modeling/emft/henshin/updates/release/>
- ▷ EGit: Eclipse Marketplace (already supplied with Eclipse Java EE Developer Release)
- ▷ Maven Integration for Eclipse: Eclipse Marketplace (already supplied with Eclipse Java EE Developer Release)
- ▷ Eclipse RCP Plug-in Developer Resources (adds plugins necessary for defining extension points in Eclipse PDE):
<http://download.eclipse.org/eclipse/updates/4.4> Version Tiger: Eclipse Marketplace
- ▷ EMF: <http://download.eclipse.org/modeling/emf/emf/updates/> OR <http://download.eclipse.org/modeling/emf/emf/updates/releases/> (Do not use the old aggregation update site <http://download.eclipse.org/modeling/emf/updates/releases/>)
- ▷ Sirius: Eclipse Marketplace
- ▷ Swtbot: <http://download.eclipse.org/technology/swtbot/releases/latest/>
- ▷ Maven Development Tools (adds lifecycle needed by Maven Plugin project): Eclipse Marketplace
- ▷ OCL: Eclipse Marketplace

After installing the required plugins we need to import the Co Wolf project into eclipse. We do this by using the dialog Import → Projects from git → Existing local repository use the path to the cloned Co Wolf project and import it. After importing the project we need to install the Ecore models of Co Wolf into eclipse, without installing the models maven is not able to generate the Henshin rules while compiling. To do this we will export the following models with the Export → Deployable plug-ins and fragments dialog.

- ▷ `de.uni_stuttgart.iste.cowolf.model.activity_diagram.feature`
- ▷ `de.uni_stuttgart.iste.cowolf.model.component_diagram.feature`
- ▷ `de.uni_stuttgart.iste.cowolf.model.ctmc.feature`
- ▷ `de.uni_stuttgart.iste.cowolf.model.dtmc.feature`
- ▷ `de.uni_stuttgart.iste.cowolf.model.fault_tree.feature`
- ▷ `de.uni_stuttgart.iste.cowolf.model.feature`

- ▷ `de.uni-stuttgart.iste.cowolf.model.lqn.feature`
- ▷ `de.uni-stuttgart.iste.cowolf.model.statechart.feature`
- ▷ `de.uni-stuttgart.iste.cowolf.core.feature`

On the destination tab use the “Install into host. Repository” option. Click finish. Now we have successfully installed the models into the Eclipse IDE. Next we will configure the target platform to enable logging.

Setting up the Logging Dependencies

For logging messages and exceptions we use the Logging framework Logback in the recommended way with the Simple Logging Facade for Java (SLF4J). In every class that uses logging a Logger object must be retrieved. For reducing the typing effort an appropriate Eclipse Code Template will be defined.

- ▷ Open the Eclipse preferences (Window → Preferences).
- ▷ choose Java → Editor → Templates and click on "New...".
- ▷ Define as name "log", as context "Java" and as Pattern the following:


```

$:import(org.slf4j.LoggerFactory, org.slf4j.Logger)
private final static Logger LOGGER = LoggerFactory.getLogger($enclosing_type)

```

At last we can build the source code by selecting the `de.uni-stuttgart.iste.cowolf.parent` project and clicking the Run As → Maven build Next set the Goals to “clean verify” and click on Run. This setup will compile the source code.

To execute the application select the `de.uni-stuttgart.iste.cowolf.ui` project and select Run As → Run Configuration → Eclipse Application. Next we need to select the plugins tab. On the plugins tab remove the following plug-ins from your CoWolf run configuration, otherwise Logback will not be initialized correctly.

- ▷ all plug-ins containing m2e
- ▷ all plug-ins containing slf4j except slf4j.api (1.7.7)
- ▷ all plug-ins containing logback except ch.qos.logback.classic (1.1.2) and ch.qos.logback.core (1.1.2)

When you click Run the CoWolf application will start.

13. Use of Co Wolf

13.2 Create New Models

Author: Verena Käfer

To create a new model you first have to create a new CoWolf project via *right-click* → *New* → *Project* → *CoWolf* → *CoWolf Project* as you can see in Figure 13.1

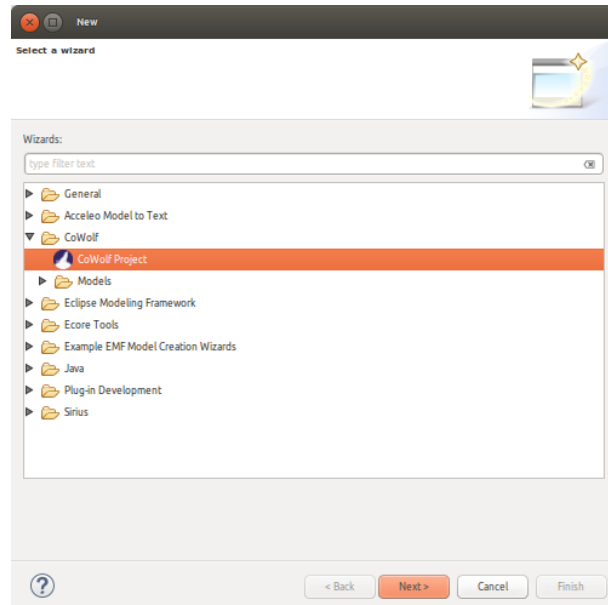


Figure 13.1. Create a new CoWolf project

Right-click on the project or folder where you want to create your new model and select *New* → *Other* → *CoWolf* → *Models*. Then select the model you want to create as it can be seen in Figure 13.2.

In the model creation wizard you can choose the parent folder and the root element of the model. You can also decide if you want to have to create a graphical representation file as well.

After creating a model with a graphical representation, the representation opens automatically. On the right side you can see the elements which can be added to the diagram in the middle. New elements can be added to the model by clicking an element icon on the right side and afterwards clicking the position where it should be created. To add a connection between two elements, click the connection icon on the right, click on the source element and then on the target element. To create several elements of the same type just press *Ctrl* during the process.

13.2. Create New Models

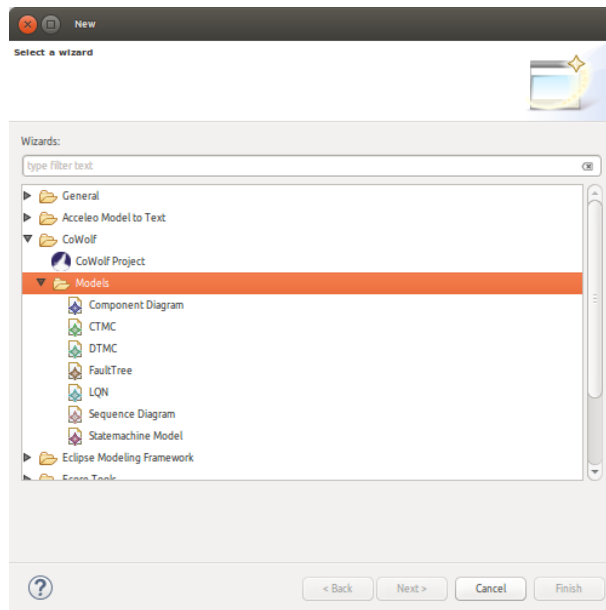


Figure 13.2. Create a new model

To edit the labels of an element, select the label, click once and then you can edit it. Some properties cannot be edited in the editor. To edit them, open the properties view, selected the element in the editor and then you can edit all properties as you wish.

To reopen a graphical editor, you need to find the aird file with the same name as your model file. In Figure 13.3 you can see the structure for a component diagram and the matching aird file. The aird file contains a folder with the representation and a link to the model file. Open the folder and expand it as far as possible. Now you can see the representation for your model. Double-click it and the editor will open. For LQN diagrams please have a look at Section 13.2.1.

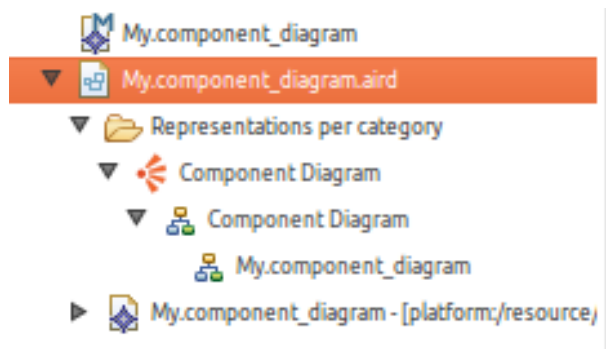


Figure 13.3. The structure of an aird file

13. Use of Co Wolf

13.2.1 What to do when...

This section describes some specialities with the graphical editors. The following points may occasionally occur.

- ▷ You try to open the graphical editor but an error message is displayed like in Figure 13.4.

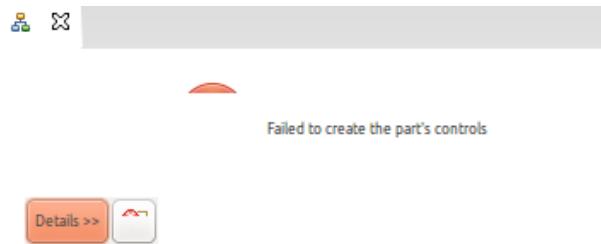


Figure 13.4. An error occurs while opening an editor

This happens mostly when opening an LQN editor after the first start of Eclipse. This is caused by a known bug in Sirius. A workaround to open the editor is to delete the according .aird file and create a new one by right-clicking on the model file and selecting Create Graphical Representation.

- ▷ You rename or move a model and the error shown in Figure 13.5 shows up:

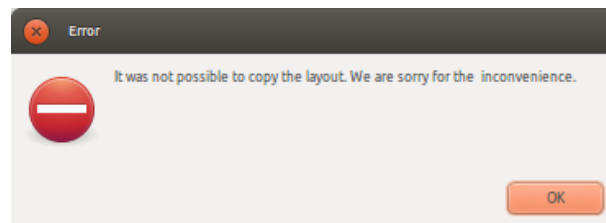


Figure 13.5. An error occurs while renaming or moving model files

This error occurs when the connection between the .aird file and the model file was lost during a move or a rename. The connection will be restored automatically, but in some cases the layout of the model will be reset.

- ▷ You move, rename or delete model files and a dialogue like in Figure 13.6 appears:

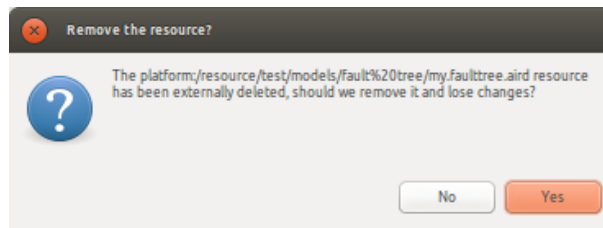


Figure 13.6. This dialogue shows up after editing model files

When you move, rename or delete a model file, the according .aird file is also changed. This dialogue asks you if you want to close the old version of the .aird file. Click Yes to confirm. If you click No, there will be an old version of the .aird file which cannot be opened any more as the according model file changed.

- ▷ You changed something in the properties view of an LQN editor and the changes are not saved. This is also a known Sirius bug and only occurs for LQN models. Changing an attribute that is not visualized in the graphical editor does not trigger a model changed event. A workaround is to perform a small change, for example repositioning an element, in the graphical editor and then saving the model.

13.3 Export Models

Author: Tim Sanwald

Co Wolf supports the export of CTMCs and DTMCs to the PRISM model language. Currently verification and automated simulation are implemented directly in Co Wolf. PRISM further supports the possibility to perform step-by-step simulation which is useful for debugging models. PRISM also provides so called "experiments" as another analysis method. Most of the users don't need these functionalities which is why they're not supported by Co Wolf directly. Other users can export the model with the implemented export method to create a PRISM model. This makes it possible to use the model with PRISM directly, so the whole functionality can be used. In the following the process to export a CTMC or DTMC is explained:

1. Open the export menu through by right clicking in the project explorer and choosing "Export...".
2. Choose "CTMC to PRISM" or "DTMC to PRISM" which opens the model export wizard (cf. Figure 13.7).
3. With the wizard it is possible to define the models which should be exported. The export destination can be selected and the export of PCTLs can be enabled.
4. By clicking on the finish button to export the model to the specified destination as a PRISM model for using it in PRISM.

13. Use of Co Wolf

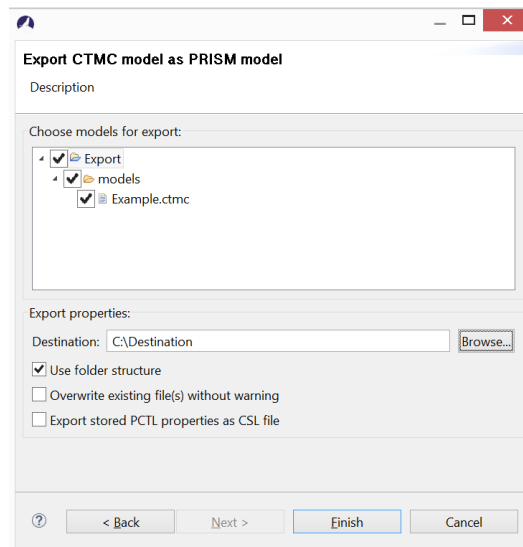


Figure 13.7. Wizard to export a CTMCs to a PRISM model.

13.4 Working with Versions

Author: Philipp Niethammer

CoWolf uses a simple version system to save and access states of a model instance that are needed for co-evolution in the future. These versions are created automatically when they are needed. Hence, a user can use CoWolf without the need to know about versions at all. Nonetheless, CoWolf contains a set of features that make use of this system.

The most important of these functionalities is the possibility to inspect the evolution of model instances over time. That is why we dedicated a separate section to it (Section 13.5).

In addition to this, a user can create a version manually at any time to save the current state of the model instance by selecting CoWolf → Versions → Create Version in the file's context menu in the Project Explorer. The user can enter an optional message to describe the new version.

It is also possible to revert the working copy to an older version. After selecting CoWolf → Versions → Revert in the file's context menu in the Project Explorer, the user is asked to select the version to revert to. The model instance is then set to the state of the selected version. This also creates a new version with the message "Reverted to version DATE.", where DATE is the date of the selected version.

To allow comfortable sharing of changes in a model instance with other team members or between machines, it is possible to create a patch that contains the changes between two versions. One way to do that is to run a difference calculation between two specific versions as described in Section 13.5 and select the option "Save difference as patch" in the

13.4. Working with Versions

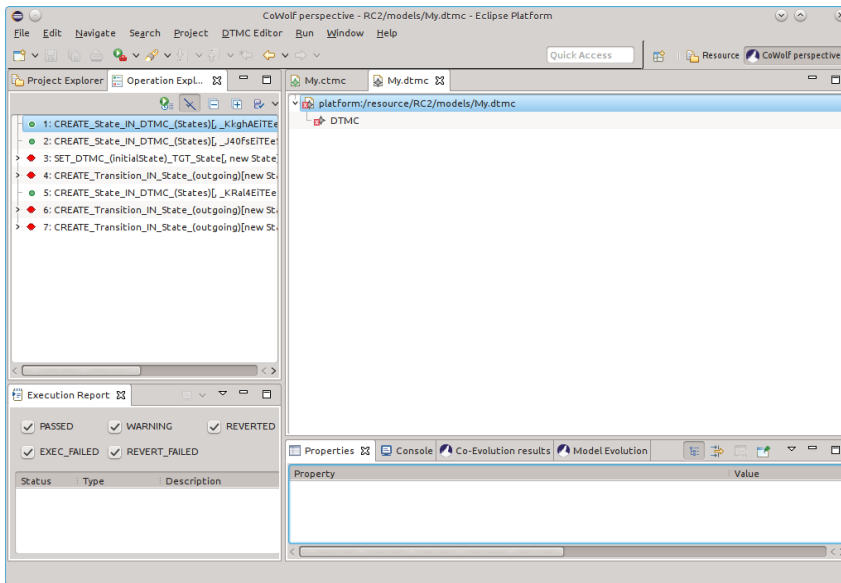


Figure 13.8. Before applying a patch, the user is shown the destined changes and asked for confirmation.

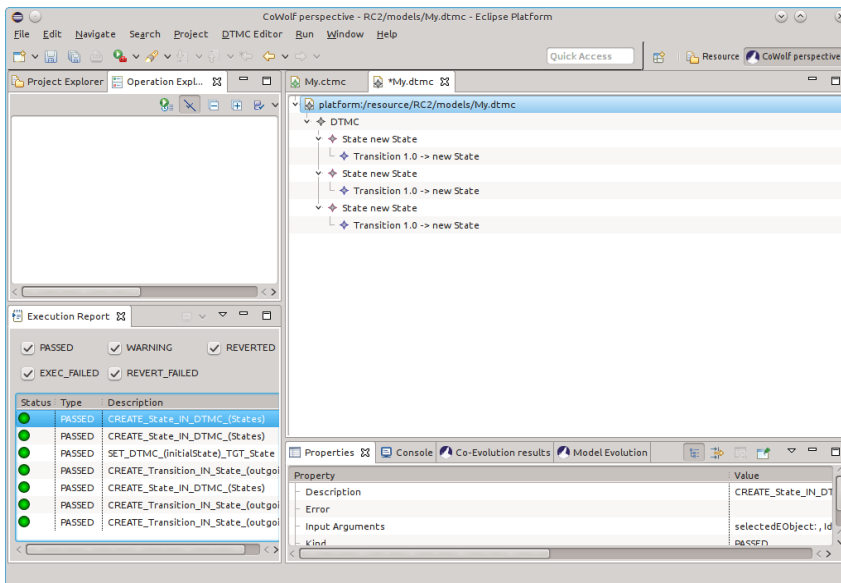



Figure 13.9. After applying a patch, the user is shown the conducted changes in detail and the resulting model.

13. Use of Co Wolf

dialogue as visible in Figure 13.10. Alternatively, it can be created by calling CoWolf → Versions → Create patch in the file's context menu which will open a similar dialogue. Either way, the user can choose a directory to save the patch file in. The file's directory is selected as default value.

To apply an existing patch file to a model instance, the user calls CoWolf → Versions → Apply patch and selects the patch file from the file system. After the patch is processed, a special view is opened, showing the target model instance, the changes included in the patch and which of these changes can or cannot be applied, as shown in Figure 13.8. The user has to start the actual modification of the model instance by pressing . After the execution, the model instance is edited and the view shows information about the process (Figure 13.9).

13.5 Evolution of a Model

Author: Michael Zimmermann

There are two ways showing for the evolution of a model. Depending on the users purpose, he can choose one of them:

▷ **Showing only the difference between two specific model versions.**

The user has to right click on the model for which he wants to see the difference between two model versions. He then has to go to CoWolf → Versions and left click on Show differences. This will open the Model Difference Wizard (see Figure 13.10). Here the user can specify the two model versions he wants so see the difference of. On the left side he has to choose which version should be the base version and accordingly on the right side of the wizard he has to choose the target version of the model. In the provided example the resulting difference will show the modifications on the model from the previously executed co-evolution (Co-evolution from models/My.ctmc). It is also possible to save the calculated difference as patch that can be applied to another model version if needed.

Figure 13.11 shows the resulting difference of the previous example. The main entries like "CREATE_State_IN_DTMC_(States)" are the change sets lifted by SiLift and represent user edit operations. The child elements of these change sets are the low-level or technical differences of the two model versions calculated by SiDiff. In the properties view at the bottom of the image you can see the values of changed attributes.

▷ **Showing the differences between all model versions and therefore the entire model evolution.**

The user has to right click on the model for which he wants to see the complete evolution. He then has to go to CoWolf and left click on Show Evolution. This will open the Model Evolution view. Figure 13.12 shows the complete evolution history for the example

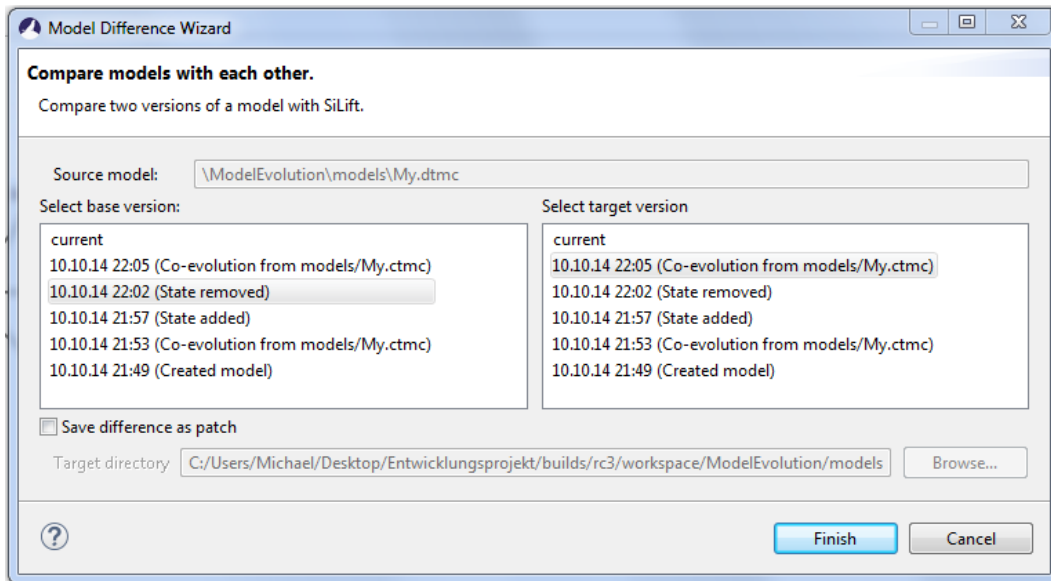


Figure 13.10. Model Difference Wizard of CoWolf. Here the user can specify the two model versions he wants so see the difference of.

model. The user can easily retrace the entire development cycle and check when the model was changed and what was changed in detail.

13.6 Co-Evolve a Model

Author: Rene Trefft

The source and target models which participate in the co-evolution must be contained in the same CoWolf-project. Further ensure that the source model is valid, otherwise the co-evolution can't be executed. Right click on the source model (not the graphical representation file `aird`) and select `CoWolf` → `Co-Evolve` in the context menu. In the appearing wizard (see `img. 13.13`) select the desired target models.

Note, only those models are shown for which a co-evolution is supported from the source model. To finally perform the co-evolution from the source model to the selected target models, click on `Finish`. During the execution the current progress is shown. When the co-evolution is finished the view `Co-Evolution results` (see `img. 13.14`) appears and shows which changes were applied to the target models.

Source and target models are linked after the execution. If the source model changes, a warning is shown in the `Problems` view for every linked file (see `img. 13.15`) to remember the user to perform a co-evolution again.

13. Use of Co Wolf

The screenshot shows a comparison window titled "Compare: 1412971328850.version <-> 1412971514128.version". The tree view shows the following structure:

- "SET_Transition_Prob"
 - Attribute-Value-Change
 - "SET_Transition_Prob"
- "CREATE_State_IN_DTMC_(States)"
 - Add-Object: New State (State)
 - Add-Reference: States (EReference) (DTMC) -> New State (State)
- "CREATE_Transition_IN_State_(outgoing)"
 - Add-Object: Transition._vzCrwIC4EeSQRZyizllhGA
 - Add-Reference: from (EReference) (Transition._vzCrwIC4EeSQRZyizllhGA -> StartState (State))
 - Add-Reference: incoming (EReference) (New State (State) -> Transition._vzCrwIC4EeSQRZyizllhGA)
 - Add-Reference: outgoing (EReference) (StartState (State) -> Transition._vzCrwIC4EeSQRZyizllhGA)
 - Add-Reference: to (EReference) (Transition._vzCrwIC4EeSQRZyizllhGA -> New State (State))
- "CREATE_Transition_IN_State_(outgoing)"
- "CREATE_Transition_IN_State_(outgoing)"
- Correspondences (15)

At the bottom, the Properties table is shown:

Property	Value
Object In Model A	Transition 0.5 -> S1
Object In Model B	Transition 0.45 -> S1
Type	prob : EFloat

Figure 13.11. Calculated difference of two model versions.

13.6. Co-Evolve a Model

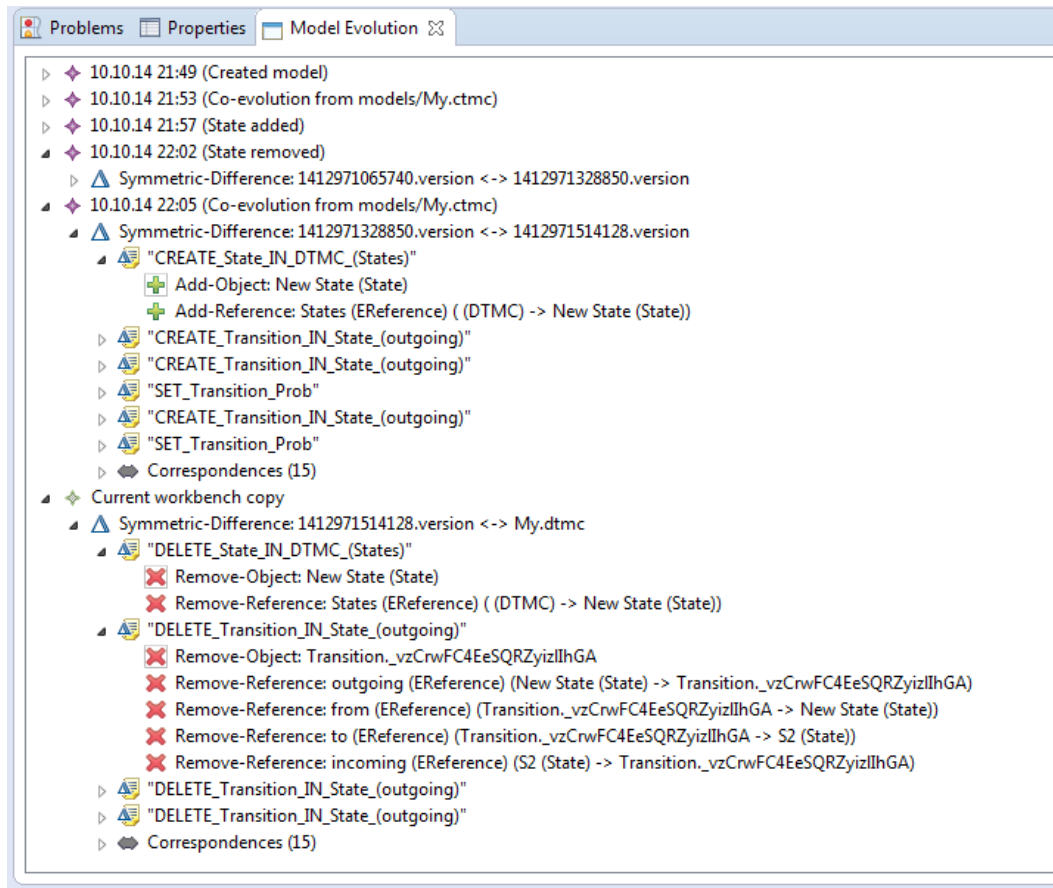


Figure 13.12. Evolution view of an example model.

13. Use of Co Wolf

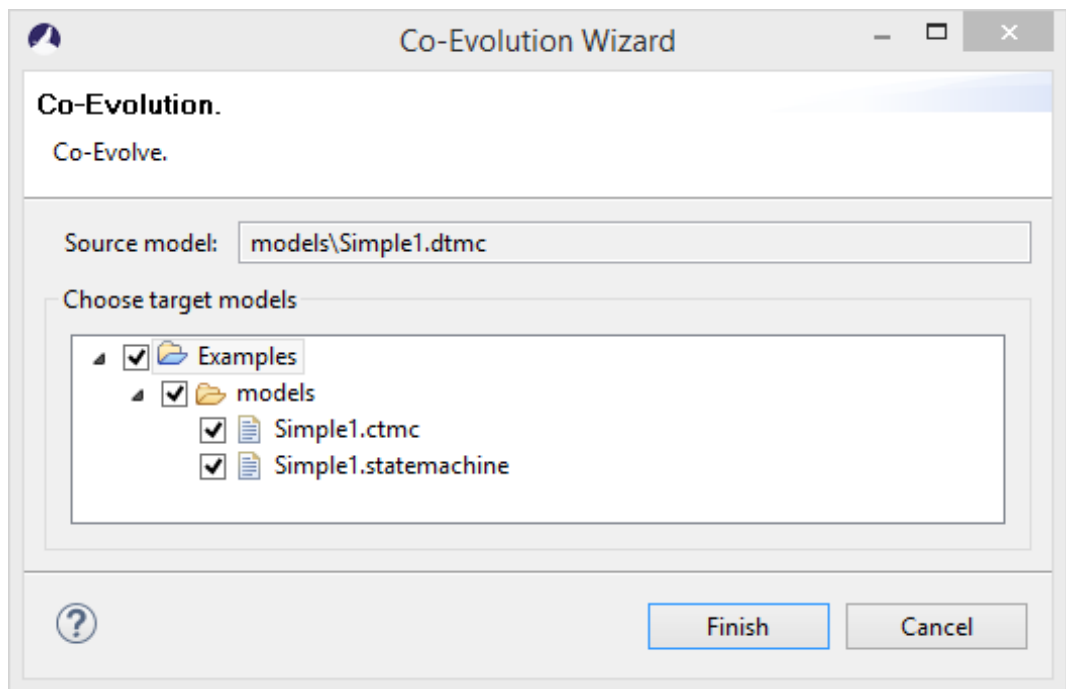


Figure 13.13. Co-Evolution Wizard of CoWolf. Here the user can specify the target models of the co-evolution. In this case the DTMC diagram will be co-evolved to a CTMC and a Statechart diagram.

13.6. Co-Evolve a Model

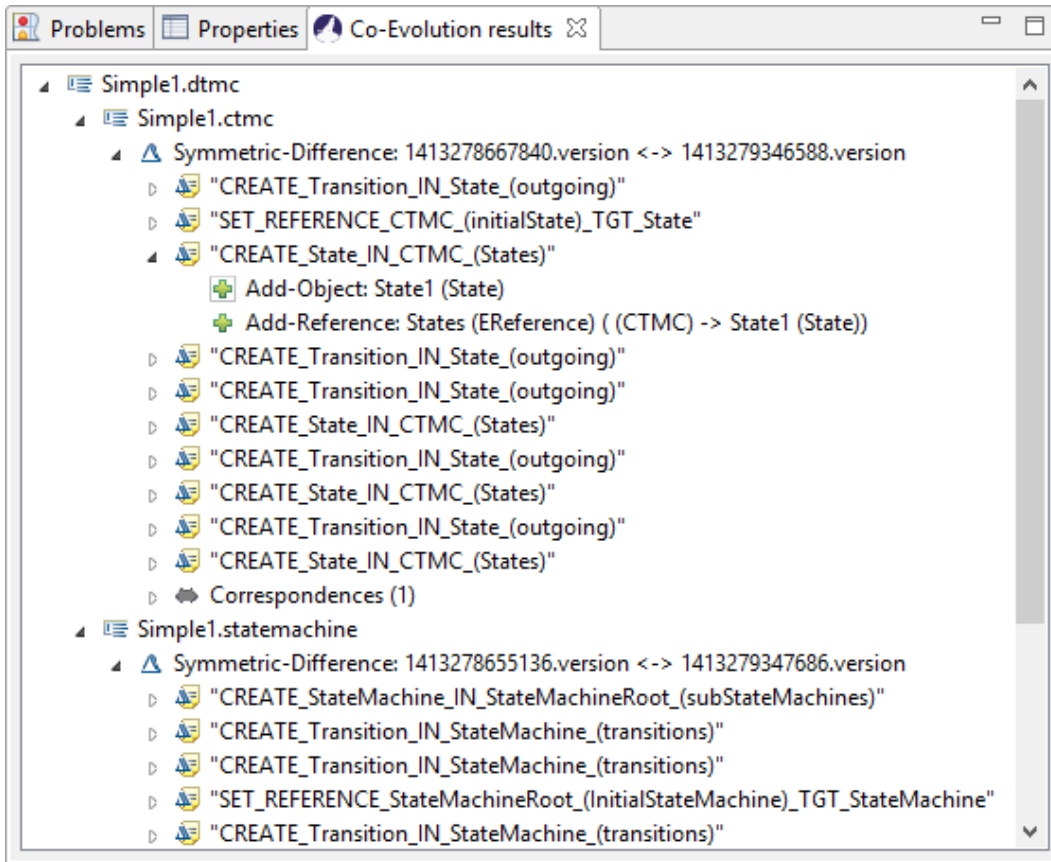


Figure 13.14. Co-Evolution Results View of CoWolf. Here the user can see the lifted change sets (with their low level changes) which were applied on the target models of the co-evolution. For example on the CTMC diagram the state creation change set was applied which consists of an add-object and an add-reference low level change.

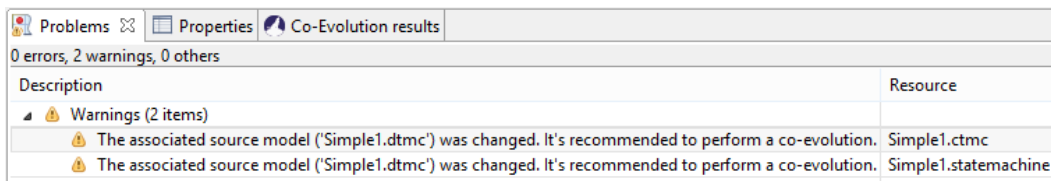


Figure 13.15. If the source model changes after a co-evolution, warnings are shown in the Problems view for every linked model.

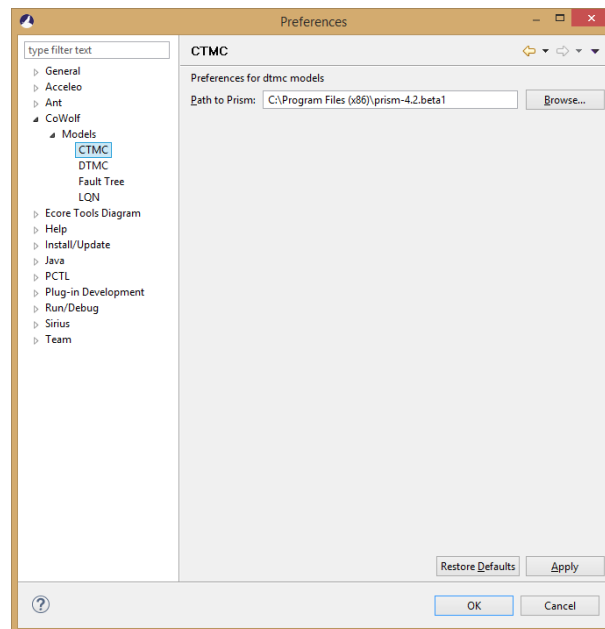
13. Use of Co Wolf

13.7 Analyze a Model

Author: David Steinhart

CoWolf has four different quality of service models, which are DTMC, CTMC, LQN and Faulttree models. These four model types can be analyzed using the corresponding tools. For DTMC and CTMC analysis, PRISM [Pri] is needed. For LQN models, the LQNS [Lqn] tool is required. And for Faulttree models, xFTA [Xft] is required.

The concept for analyzing a model is the same for all model types. The first step is to make sure that the required tools have been installed and the path to each of these tools has been set in the properties. To set the path, select Window→Preferences to get to the Preferences view. Select the entry labeled CoWolf→Models on the left hand side and select the required model there.



When

Figure 13.16. Preferences view for CTMC

the path to the required tool has been set, you need to make sure that the model you want to evaluate is valid. To make sure that a model is valid, open the graphical representation file of the model, right-click and select "Validate diagram". You can also open the tree view editor, right-click the top node of the model and click "Validate". It is important to click the top node in the tree view editor, otherwise only the selected element and all of its children will be validated. If a model is not valid, any errors found will be shown in the Problems window as well as in the graphical and the tree view editor. Validating the models is necessary, otherwise the analysis tool might either throw an exception or return wrong results.

To start the analysis, right-click the model you want to analyze and select CoWolf→Analyze. A different wizard will be shown for each model. When the analysis has been executed successfully, the results are saved to an HTML-document.

LQN

The LQN analysis wizard provides only one option, which is to solve the selected LQN model. Select Finish to perform the analysis.

Fault Tree

The Faulttree analysis wizard has two options. It is possible to calculate the probability of the top event, which indicates how likely the top hazard will occur. It is also possible to calculate a minimal cutset to find all possible combinations which will trigger the top hazard. Select the required option, then select Finish to perform the analysis.

DTMC

The DTMC analysis wizard can be used to select states for which a reachability analysis should be performed. PRISM supports two kinds of reachability analyses, verification and simulation. Verification analyses the behavior of the DTMC when an infinite number of steps would be performed. The results are therefore always the same and evenly distributed. Simulation runs an experiment with a given number of samples, a maximum path length and a required confidence. For simulations, results vary for each run and for each configuration. After choosing either verification or simulation, the states and labels that should be analyzed have to be selected. There is also the option to include all absorbing states, which are all states that do not have any outgoing transitions. When all required states and labels have been selected, click Finish to run the analysis.

CTMC

The CTMC analysis wizard can be used to perform basic reachability and performance analysis, but can also be customized to analyze a wide range of properties. The CTMC analysis wizard contains a list of all properties that have been created so far. Check the box on the left hand side of an existing property if you want to analyze it. Select Finish to analyze all selected properties. To add a property, click the Add-button on the bottom right. To edit or delete a property, highlight it in the list and then click the Edit-button or the Delete-Button. When a property is added or edited, the property wizard will be opened.

13. Use of Co Wolf

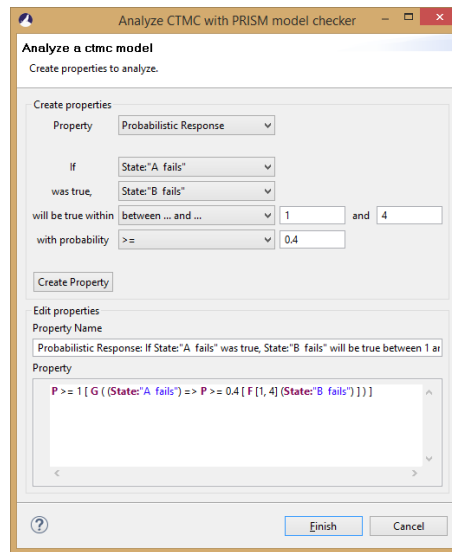


Figure 13.17. CTMC properties wizard

The property wizard contains a drop down menu at the very top which offers four different options: Steady State Probability, Probabilistic Existence, Probabilistic Until and Probabilistic Response. When selecting one of the properties, the corresponding fields in the top area will become editable. The texts on the left hand side and the drop down menus next to them form a sentence that explains what can be analyzed with the current property. When all required fields are filled in and "Create Property" is clicked, the corresponding PCTL definition of the property is calculated and inserted in the Xtext text field at the bottom. Additionally, a suggestion for a meaningful title is created. As some users may be interested in analyzing additional, more complex properties, the Xtext text field at the bottom can be edited. It is possible to create PCTL expressions, which will be validated right away and errors will be highlighted. The Xtext text field contains auto-completion using Ctrl+Space. Boolean Expressions can be nested using and (&), or (|), not (!), true (true) and false (false) and need to be in brackets. States and Labels need to follow the form 'State:"State Name"' and 'Label:"Label Name"'.

Future Work

During the work on CoWolf we realized that there are a few things that would make the usage of CoWolf easier but could not be done yet. They are described here.

14.1 Development

Author: Verena Käfer

There are some approaches which could make the development easier. These are described in the following section.

14.1.1 Static Analysis on Rules for Automatic Prioritization

During a transformation between two models, the used transformation rules are executed in a specific order. For example must a state always be created before an attached label can be created in this state. To do this every rule has a manually set priority. This can lead to errors, as it is possible that after the creation of a new rule to simply forget to redo the prioritization.

A much better solution would be a static analysis of the rules to decide on the fly which rule has to be used first in each case. This could prevent human errors.

14.1.2 Iterative Sample-based Rule Generation

At the moment all transformation rules between two models have to be done by hand. It would be nice to be able to create the rules automatically. One approach could be to find a small rule base from samples of corresponding models and then iteratively extend this rule base by analysing further samples or manual co-evolution.

14.2 Functionality

Author: Verena Käfer and Philipp Niethammer

This section describes features and functionality for the CoWolf program on the end user side. In difference to the last section about the development of new features, it concentrates on what could improve the user experience of CoWolf.

14. Future Work

14.2.1 Multiple Analysis Methods

In the current architecture, the analysis is a fixed part of a QoS model plugin. Because of that, it is currently not possible to provide the model analysis separately from the model or to provide multiple methods to analyze a model instance. Though, this might be a very interesting feature if different tools allow for different analysis techniques or, for example, to integrate a commercial tool.

14.2.2 Multiple Transformation Providers

In the current version of *CoWolf* every transformation has exactly one set of rules which is executed. There might be situations in which it is necessary to have different transformation rules between two models for different results, for example to have different transformed elements for an analysis. This would mean that the user could choose which rule base to take before a transformation.

14.2.3 Synchronization between two Model Instances

As an extension to the unidirectional co-evolution between models, a synchronization of changes in models discovers changes on both sides (A and B) simultaneously and separates them in three sets: changes that are made solely in model A, changes solely in model B and last, changes that are made in both models, either conflicting or corresponding.

For example, a user may create the state named "MyNewState" in both, the State Chart and the associated DTMC. The synchronization detects this corresponding change and adds a trace between the created states to save it. On the other hand, existing, associated states could have been renamed differently. With simple serial co-evolution, the order of execution is important as the first direction overwrites the changes in the target model and thus leads to a lost update. With synchronization, the changes are detected as conflicting and a solution can be found, e.g. by letting the user decide to skip this transformation or choose the applied direction.

14.2.4 Automatic Co-Evolution and Synchronization

Currently when the user makes changes in a model instance, a warning is shown on all associated models that they should be updated, but the user has to start the co-evolution himself. To improve the user experience, the co-evolution could be executed automatically. For example, after the user makes a change in a component diagram, this change is immediately propagated to corresponding fault trees and to DTMC instances associated with any of these fault trees afterwards.

While the automatic execution is relatively easy as we can find all associated model instances using the `ModelAssociationManager`, we see three major obstacles:

First, before we can execute the automatic co-evolution, the order of execution must be planned and cycles identified and solved. Cycles can be either direct, when the transformation is supported bi-directional, or indirect, running over several instances. On one hand, undetected cycles can lead to an infinite loop and must be prevented at all costs, on the other hand, a suboptimal solution could stop the process too early so that not all changes are propagated completely.

Second, a transformation is only possible if the source model is valid. Often, model instances are not valid after transformation and the user has to complete them manually. The hierarchical co-evolution would either abort here - maybe invalidating the execution plan - or interact with the user to complete the information on the fly. This might be unwanted by the user.

Third, automatic co-evolution is mainly a matter of performance. According to our evaluation (cf. Chapter 11), a single co-evolution takes at least about 10 seconds at the moment, even for a small change set in a relatively small model. This may become quite annoying for the user, if it automatically happens too often, e.g. on every save. This could be relieved by triggering it less often, what can in turn jeopardize the idea of automatic execution.

14.2.5 Work with Delta Sets

In our evaluation (Chapter 11) we showed, that co-evolution is normally much faster than simple transformation for big models. Admittedly, the duration of the co-evolution is not independent of the size of the model instance, though. During the transformation process, Henshin searches for a specific pattern of objects and dependencies in the transformation graph. This graph contains of three model instances, two sets of traces and the difference information, as described in Section 12.3.2. Obviously, with increasing size of model instances, and hence graph, the pattern matching process slows down.

But the greater part of the source model instances might not even be used by the transformation. By analyzing the transformation rules that are to be executed, unnecessary parts of the instances could be filtered before execute the transformation.

On the target side, one could think about generating a change set instead of modifying the target instance directly. Thus, the target side is empty in the beginning and its size only depends on the size of changes, not the size of instances. After the transformation, the target instance is modified en bloc according to the change set.

14.2.6 Headless Operation

With increasing size of models and therefore increasing duration of operations like co-evolution or analysis, a common working station computer will come to its limits. As it is already common practice in software development or video editing, a dedicated work machine or cluster could be used to accelerate the execution. This requires a headless operation of these operations, that is, the direct execution from, e.g., a command line

14. Future Work

without the need of a GUI or a complex infrastructure. Achieving that, a company could for example run nightly jobs to transform models in an analyzable form, analyze it afterwards and integrate the results in their quality management and continuous integration system in that way.

14.2.7 Merged Model Editor

To see the connection between two or more models it could be useful to open several models and their traces in one editor. The connection between the models would be clearly visible and it would also be possible to refactor the models, for example to reset traces or to create new traces.

14.2.8 Model Annotation Infrastructure

Showing the results of an analysis directly in the model instance is already a feature that is asked for. While it is an easy task to extend the models for fields holding the analysis result and writing to these fields after analysis, this increases the complexity of the models and is not flexible when using different analysis methods that results in different types of results. Promising instead is the idea of creating a generic infrastructure to annotate instances with all kinds of data and referencing services in these annotations that describe the further usage of the data. In this way, for example, an analysis method could describe how the data is displayed in the editor and even, how it is transformed for another model type. For example, the CTMC analysis method could not only define how the analysis results are annotated to the CTMC model instance, but also how these results can be transformed to be annotated to the associated fault tree.

Conclusion

Author: Verena Käfer

As stated in the introduction, models are important in today's software systems. However, a classic transformation between two models takes too long and loses too much information. To increase this, CoWolf uses an incremental transformation approach. It supports seven different models and the transformations between pairs of them as well as the analysis of some of them.

For CoWolf we had three main goals explained in the introduction and it can be said, that we reached all of them:

- ▷ The management of associations between Model Instances
- ▷ Deliver utilities for model development and analysis
- ▷ The co-evolution of an associated model on the basis of evolutions

CoWolf stores the associations between different model instances and also reminds you of a new co-evolution if a model has changed. It provides an easy user interface with graphical editors for every model as well as wizards for the analysis of the quality of service models. Most important, it can do an incremental transformation between two models.

To do such an incremental transformation we first need the difference between the current version and previous version of a model. This difference is mapped on Henshin rules which indicate what has to be changed in the second model. Then the changes are applied.

This leads to two main advantages. First, the time for applying the changes is much faster for an incremental transformation as for a complete transformation. Second, the adoptions the user has to make after the transformations are less for the incremental approach because the changes the user already made are not lost. This saves a lot of time. We evaluated both, the faster transformations and the smaller amount of adoptions, in a case study.

Of course there are some things left for future work but all in all CoWolf includes everything we wanted to do. It is a useful framework for incremental model transformation which can save a lot of time and work as it is open source and extensible. There are no borders on making improvements for new models or analyses. In our opinion it is a step in the right direction to do incremental transformations between models and CoWolf supports this.

Bibliography

- [Are+10] Thorsten Arendt et al. “Henshin: advanced concepts and tools for in-place emf model transformations”. In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I. MODELS’10*. Oslo, Norway: Springer-Verlag, 2010, pp. 121–135.
- [Bar07] Neil Bartlett. A Comparison of Eclipse Extensions and OSGi Services. Feb. 2007. URL: <http://www.eclipsezone.com/articles/extensions-vs-services>.
- [BCS07] H. Boudali, P. Crouzen, and M. Stoelinga. “Dynamic fault tree analysis using input/output interactive markov chains”. In: *Dependable Systems and Networks, 2007. DSN ’07. 37th Annual IEEE/IFIP International Conference on*. June 2007, pp. 708–717.
- [BET10] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. “Lifting parallel graph transformation concepts to model transformation based on the eclipse modeling framework”. In: *ECEASST 26* (2010).
- [Bie+06] Enrico Biermann et al. “Graphical definition of in-place transformations in the eclipse modeling framework”. In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems. MODELS’06*. Genova, Italy: Springer-Verlag, 2006, pp. 425–439.
- [Bor07] Artur Boronat. “Moment: a formal framework for model management”. In: *PhD in Computer Science, Universitat Politècnica de València (UPV), Spain* (2007).
- [BP08] Cédric Brun and Alfonso Pierantonio. “Model differences in the eclipse modeling framework”. In: *UPGRADE, The European Journal for the Informatics Professional* (2008).
- [Bra11] MGJ Brand. “Rcvdiff - a stand-alone tool for representation, calculation and visualization of model differences”. In: *ME 2010 International Workshop on Models and Evolution (Oslo, Norway, October 3, 2010; co located with ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems)* (2011).
- [BV06] András Balogh and Dániel Varró. “Advanced model transformation language constructs in the viatra2 framework”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing. SAC ’06*. Dijon, France: ACM, 2006, pp. 1280–1287.
- [CDMI11] Vittoria Cortellessa, Antinisca Di Marco, and Paola Inveradi. *Model-Based Software Performance Analysis*. Springer, 2011.

Bibliography

- [Cha01] Scott Chacon. GitHub Flow. 2011. URL: <http://scottchacon.com/2011/08/31/github-flow.html>.
- [CKB14] University of Stuttgart Christian Karl Bernasko. Seminar paper: Continuous delivery and Continuous integration. Apr. 2014.
- [DN93] Lorenzo Donatiello and Randolph Nelson. Performance Evaluation of Computer and Communication Systems. Vol. 729. Lecture Notes in Computer Science. Springer, 1993.
- [Ehr+06] H. Ehrig et al. Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- [Emf] Eclipse Modeling Framework project page. URL: <http://www.eclipse.org/modeling/emf/>.
- [ER08] Steven Epstein and Antoine Rauzy. Open-PSA Model Exchange Format. 2008. URL: <http://www.open-psa.org/>.
- [Fra+13] Roy Gregory Franks et al. Layered Queueing Network Solver and Simulator User Manual. 2013. URL: <http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf>.
- [Fra99] Roy Gregory Franks. "Performance Analysis of Distributed Server Systems". PhD thesis. Carleton University, 1999.
- [Gal] Robert Gallager. Finite-State Markov Chains. URL: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-262-discrete-stochastic-processes-spring-2011/course-notes/MIT6_262S11_chap03.pdf.
- [Gar+07] Hubert Garavel et al. "CADP 2006: A toolbox for the construction and analysis of distributed processes". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, pp. 158–163.
- [GHS09] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. "Improved flexibility and scalability by interpreting story diagrams". In: *ECEASST 18 (2009)*.
- [GPH14] Ceki Gülcü, Sébastien Pennec, and Carl Harris. Logback Manual. 2014. URL: <http://logback.qos.ch/manual>.
- [HF10] Jez Humble and David Farley. Continuous Delivery - Reliable Software Releases Through Build, Test, and Deployment Automation. 1. Aufl. Amsterdam: Addison-Wesley, 2010.
- [HK10] Markus Herrmannsdoerfer and Maximilian Koegel. "Towards a generic operation recorder for model evolution". In: *Proceedings of the 1st International Workshop on Model Comparison in Practice. IWMCP '10*. New York, NY, USA: ACM, 2010, pp. 76–81.

- [JE04] Sven Johann and Alexander Egyed. "Instant and incremental transformation of models". In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 362–365.
- [Keh+14a] Timo Kehrer et al. SiLift. 2014. URL: <http://pi.informatik.uni-siegen.de/Projekte/SiLift/>.
- [Keh+14b] Timo Kehrer et al. SiLift: Tool Environment Overview. 2014. URL: <http://pi.informatik.uni-siegen.de/Projekte/SiLift/overview.php>.
- [Ker] Kermeta - Website. 2014. URL: <http://kermeta.org>.
- [KNP11] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: verification of probabilistic real-time systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.
- [Kol+07] D.S. Kolovos et al. "Update transformations in the small with the epsilon wizard language". In: *Journal of Object Technology* 6.9 (2007), pp. 53–69.
- [KR06] Harmen Kastenberg and Arend Rensink. "Model checking dynamic states in groove". In: *Proceedings of the 13th International Conference on Model Checking Software*. SPIN'06. Vienna, Austria: Springer-Verlag, 2006, pp. 299–305.
- [Kus00] Sabine Kuske. "'Transformation Units – A structuring Principle for Graph Transformation Systems'". PhD thesis. University of Bremen, 2000.
- [Kög08] Maximilian Kögel. "TIME - tracking intra- and inter-model evolution". In: *Software Engineering 2008 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 18.-22.2.2008 in München*. 2008, pp. 157–164.
- [Kön10] Patrick Könemann. "Capturing the intention of model changes". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 108–122.
- [Ley13] Frank Leymann. Model Driven Architecture. University Lecture. 2013.
- [LFVH13] Christoph Legat, Jens Folmer, and Birgit Vogel-Heuser. "Evolution in industrial plant automation: a case study". In: *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*. IEEE. 2013, pp. 4386–4391.
- [LO92] Ernst Lippe and Norbert van Oosterom. "Operation-based merging". In: *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*. SDE 5. New York, NY, USA: ACM, 1992, pp. 78–87.
- [Lqn] Layered Queueing Network Solver and Simulator Website. URL: <http://www.sce.carleton.ca/rads/lqns/>.
- [MB+14] G. Blondelle M. Barbero L. Goubet et al. EMF Compare/CompareUMLPapyrusAPI. 2014. URL: http://wiki.eclipse.org/EMF_Compare/CompareUMLPapyrusAPI.

Bibliography

- [MM01] Joaquin Miller and Jishnu Mukerji. Model Driven Architecture - A Technical Perspective. 2001. URL: <http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf>.
- [Mol] MOLA - Website. 2014. URL: <http://mola.mii.lu.lv>.
- [Ocl] Object Constraint Language (OCL). URL: <http://www.omg.org/spec/OCL>.
- [OMG14] Object Management Group. Model Driven Architecture (MDA): The MDA Guide Rev 2.0. 2014. URL: <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf>.
- [Pal] Palladio. URL: https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model.
- [Pri] PRISM Website. URL: <http://www.prismmodelchecker.org>.
- [Rau12] Antoine Rauzy. XFTA Manual. 2012. URL: <http://www.lix.polytechnique.fr/~rauzy/xfta/XFTA-Manual.pdf>.
- [Rin14] Michaela Rindt. SERGe - SiDiff EditRule Generator. 2014. URL: <http://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/SERGe.php>.
- [Roy70] Winston W Royce. "Managing the development of large software systems". In: *proceedings of IEEE WESCON*. Vol. 26. 8. Los Angeles. 1970.
- [Sch97] Ken Schwaber. "SCRUM Development Process". English. In: *Business Object Design and Implementation*. Springer London, 1997, pp. 117–134.
- [Sd] UML Sequence Diagrams. URL: <http://www.uml-diagrams.org>.
- [SDG09] J. Sutherland, S. Downey, and B. Granvik. "Shock therapy: a bootstrap for hyper-productive scrum". In: *Agile Conference, 2009. AGILE '09*. Aug. 2009, pp. 69–73.
- [Sha] Mehrdad Mirshams Shahshahani. Continuous Time Processes. URL: <http://web.stanford.edu/class/stat217/Chapter3.pdf>.
- [Sir] Sirius. URL: <http://www.eclipse.org/sirius/>.
- [Slf] SLF4J Manual. 2014. URL: <http://www.slf4j.org/manual.html>.
- [SS13] Ken Schwaber and Jeff Sutherland. Scrum Guide. 2013. URL: <http://www.scrumguides.org/scrum-guide.html>.
- [SS14] Ken Schwaber and Jeff Sutherland. Scrum History. 2014. URL: <http://www.scrumguides.org/history.html>.
- [SZN04] Christian Schneider, Albert Zündorf, and Jörg Niere. "Coobra – a small step for development tools to collaborative environments". In: *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering*. 2004.
- [Tae+14] Gabriele Taentzer et al. "A fundamental approach to model versioning based on graph modifications: from theory to implementation". English. In: *Software & Systems Modeling* 13.1 (2014), pp. 239–272.

Bibliography

- [UI08] Francis Upton IV. Support IResourceDelta.COPIED_FROM on IResourceChangeListener. Feb. 2008. URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=217489.
- [Umla] UML 2. URL: <http://www.eclipse.org/modeling/mdt/?project=uml2>.
- [Umlb] UML-Designer. URL: <https://github.com/0beoNetwork/UML-Designer>.
- [Vog13a] Lars Vogel. Eclipse p2 updates for RCP applications - Tutorial. June 2013. URL: <http://www.vogella.com/tutorials/EclipseP2Update/article.html>.
- [Vog13b] Lars Vogel. OSGi Modularity - Tutorial. June 2013. URL: <http://www.vogella.com/tutorials/OSGi/article.html>.
- [Vog14] Lars Vogel. Extending the Eclipse IDE - Plug-in development - Tutorial. Feb. 2014. URL: <http://www.vogella.com/tutorials/EclipsePlugIn/article.html>.
- [Xft] XFTA Website. URL: <http://www.lix.polytechnique.fr/~rauzy/xfta/xfta.htm>.
- [Xte] Xtext. URL: <http://www.eclipse.org/Xtext/>.
- [Cuc13] Cucumber Team. Gherkin. 2013. URL: <https://github.com/cucumber/cucumber/wiki/Gherkin>.
- [IBM+08] IBM Corporation et al. Resource Set (EMF Javadoc). 2008. URL: <http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org.eclipse.emf.ecore/resource/ResourceSet.html>.
- [OCL14] OCLinEcore. Oct. 2014. URL: <http://wiki.eclipse.org/OCL/OCLinEcore>.
- [Sof14a] Software Engineering Group, University of Siegen. SiDiff. 2014. URL: <http://pi.informatik.uni-siegen.de/Projekte/sidiff/index.php>.
- [Sof14b] Software Engineering Group, University of Siegen. SiLift - Benutzerhandbuch für Endanwender. Apr. 2014. URL: http://pi.informatik.uni-siegen.de/Projekte/SiLift/downloads/userguide_enduser.pdf.
- [Sof14c] Software Engineering Group, University of Siegen. The SiDiff Components. 2014. URL: pi.informatik.uni-siegen.de/Projekte/sidiff/components.php.
- [The14a] The Apache Software Foundation. Apache Maven - Website. 2014. URL: <http://maven.apache.org>.
- [The14b] The Eclipse Foundation. EMFCompare - Compare and Merge Your EMF Models. 2014. URL: <http://www.eclipse.org/emf/compare/>.

Glossary

Ecore Metamodel Part of the Eclipse Modeling Framework (EMF). A meta model for describing object oriented models using mainly packets, classes, references and attributes . 35, 36

Ecore Model The abstract definition of a model using the Ecore meta model. 29, 32, 93

Maintenance engineer A person which supports the functionality of the system after the development project. 28, 29, 34

Model Instance Data that follows the scope, structure and rules of a model. A Model Instance can change over time. This process is called evolution. This data is typically either represented as Java objects or serialized in an XML data structure . 2, 30, 32, 93, 95

Model Plugin Part of the CoWolf architecture. One or more Eclipse plugins, clustered as feature. A Model Plugin contains the meta model, the analyzes for quality of service models and one ore many editors for model instances. . 95, 144

Model Version Describes a specific state in the evolution of a model instance. 93–95

user A person which uses the deployed system. 27–34

Acronyms

CTMC Continuous Time Markov Chain. 29–31, 33, 42, 95–97, 159, 160, 168, 169

DTMC Discrete Time Markov Chain. 27, 29–31, 33, 42, 95, 96, 159, 168, 169

EMF Eclipse Modeling Framework. 35–37, 103, 134, 136

JAXB Java Architecture for XML Binding. 106

LQN Layered Queueing Network. 29, 31, 33, 55, 58, 59, 61, 74–76, 91, 92, 95, 97, 168, 169

OCL Object Constraint Language. 30, 36

PCTL Probabilistic Computation Tree Logic. 30, 159

QoS Quality of Service. 95, 172

VCS Version Control System. 22, 93, 95

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 21 October 2014,

Christian Karl Bernasko

Manuel Borja

Verena Käfer

David Krauss

Michael Müller

Philipp Niethammer

Tim Sanwald

Jonas Scheurich

David Steinhart

Rene Trefft

Johannes Wolf

Michael Zimmermann