Graph Based Verification of Software Evolution Requirements

Selim Ciraci

Graph Based Verification of Software Evolution Requirements

Selim Ciraci

Ph.D. dissertation committee:

Chairman and secretary:

Prof. Dr. Ir. A.J. Mouthaan, University of Twente, The Netherlands

Promotor:

Prof. Dr. Ir. M. Akşit, University of Twente, The Netherlands

Assistant promotor:

Dr. P. M. van den Broek, University of Twente, The Netherlands

Members:

Prof. Dr. U. Aßmann, Technical University of Dresden, Germany
Prof. Dr. J. Bézivin, University of Nantes, France
Dr. Ir. A. Rensink, University of Twente, The Netherlands
Prof. Dr. J. C. van de Pol, University of Twente, The Netherlands
Prof. Dr. J. Zhao, Shanghai Jiao Tong University, China
Prof. Dr. J. Whittle, Lancaster University, England

CTIT Ph.D. thesis series no. 09-162. Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

This work has been carried out as a part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

ISBN 978-90-365-2956-3
ISSN 1381-36-17 (CTIT Ph.D. thesis series no. 09-162)

Cover design by Selim Ciraci
Printed by PrintPartners Ipskamp, Enschede, The Netherlands

GRAPH BASED VERIFICATION OF SOFTWARE EVOLUTION REQUIREMENTS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
'prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday the 17th of December 2009 at 15.00

by

Selim Ciraci

born on the 5th of October 1981
in Ankara, Turkey

This dissertation is approved by

Prof. Dr. Ir. M. Akşit (promotor)
Dr. P. M. van den Broek (assistant promotor)

Dedicated to my family; especially to my father Salim Ciraci who believed in me from the beginning

Acknowledgements

First of all I would like thank my daily supervisor Pim van den Broek and promoter Mehmet Aksit for their support, guidance and trust in me. Working with them showed and taught me a lot about academia and research processes. Their supervision not only taught me the principles of rigid research but also motivated me for finding new challenges. More importantly, from them I have learned personal communication skills, which I lacked before I started my work at University of Twente. Besides work, we were able to talk about many things and were able to have fun under the pressure of work load. I'm also very grateful for these nice discussions. I, personally, believe that they had more trust in me than I had for myself during this PhD work. I'm mostly happy that I was able to hold my promise on getting this degree.

I am very grateful that I had a daily supervisor who was well interested in my research and who had always time for me. During my PhD work, I visited Pim's office at least two times a week. In all these "short meetings", he helped me in solving problems; not only in research related problems but also for problems concerning writing skills and personal communication.

The first years of this PhD work were rather painful as I was learning how to do research. The major drawback I had in these years was my tendency to de-motivate myself. Here, I would like thank Mehmet Aksit again for finding new ways to challenge me and motivating me to work more. Sometimes in our meetings, I was amazed by his way of thinking and, more importantly, foreseeing the road ahead.

I would like to thank to the members of my PhD committee: Uwe Aßmann, Jean Bézivin, Arend Rensink, Jaco van de Pol, Jon Whittle and Jianjun Zhao for spending time in evaluating my work. With their useful comments the quality of this thesis has been greatly improved.

I would like to thank the members of the Darwin project for their feedback during the project meetings. Particularly, the comments of Dave Watts, Pierre van de Laar and Pierre America had impacts on the direction of this research. The Darwin

project also provided me the opportunity to work with the industry. The experience I have gained with this industrial collaboration provided me with ideas about many research directions. From the Darwin industrial partner, I would like to thank Lennart Hofland for his support and help in my work. All the case studies this thesis presents are provided by him.

In this thesis, I have carried out two experiments. I think experimentation in Software Engineering is very important because we, as a research field, claim that we introduce methods/tools to ease the software development and, thus, we should do experiments/case studies to prove that these methods/tools in fact help the developers. However, designing and organizing such an experiment with a human factor is a very hard process as one needs to consider/control many aspects. Luckily, I had help while designing the experiments presented in this thesis. I would like thank Klaas van den Berg for his help in the design process and Peter Geurts for his help in the data analysis process.

I am also very lucky to meet and work with the members of the software engineering (TRESE) group. During our regular seminars they provided me with very useful feedback. In addition to this, we also had great discussions which made the daily work enjoyable. Here, I would like to thank Joost Noppen who helped me out with my "introduction" to the TRESE environment and also made quite some phone calls when I had problems with Dutch. I also would like to thank Ivan Kurtev for reading my papers, teaching me the secrets of academic writing and listening to me when I talk hours about electronic gadgets.

I would like to thank Ellen Roberts-Tieke, Joke Lammerink, Elvira Dijkhuis, Hilda Ferwerda, Nathalie van Zetten and Jeanette Rebel-de Boer for their invaluable administrative support.

The first day I have came to the Netherlands, I have met with Feridun Ay. From that day on, he personally helped with every possible problem I faced and thanks to him I got acquainted with procedures and to the environment rather easily. Feridun also introduced me to the Turkish community at the university, which later became the Turkish Student Association (TUSAT). I consider myself lucky to meet with this community, as thanks to them I never felt homesick and I have established great friendships. I would like to thank Ozlem Durmaz Incel, Gokhan Doygun, Erhan Bat, Ayse Morali, Janet Acikgoz, Aysegul Tuysuz Erman, Emre Dikmen for their support in establishing TUSAT. Especially, Hasan Sozer, the first president of TUSAT, who also supported me during the PhD work and Red Alert.

I have been sharing an apartment with Arda Goknil and Gurcan Gulesir. After a stressful day it is very important to come home and relax. Arda and Gurcan are the ultimate house mates in this manner; they are always ready to go to cinema,

talk and have fun. They are most understanding when I was busy and could not help them with the arrangements. Most importantly, since we were all hard working students, work had a priority over the "living conditions" and we did a great job in ignoring the condition of the house. Besides my house mates and the Turkish community, I had a chance to meet and establish friendship with great people. From these great people, I would like to thank Espen Hamborg (Mr. Amplifier), Suzanne Engelsman and Elske Olthof for all their help and support.

I have endless regards and love for my family for believing in me and supporting me throughout my life. I would like to thank my parents who stood by me regardless of many obstacles. I cannot express my gratitude with words...

Lastly, for all my friends and family: *May the force be with you!*.

Abstract

Due to market demands and changes in the environment, software systems have to evolve. However, the size and complexity of the current software systems make it time consuming to incorporate changes. During our collaboration with the industry, we observed that the developers spend much time on the following evolution problems: designing runtime reconfigurable software, obeying software design constraints while coping with evolution, reusing old software solutions for new evolution problems. This thesis presents 3 processes and tool suits that aid the developers/designers when tackling these problems.

The first process and tool set allow early verification of runtime reconfiguration requirements. Runtime reconfiguration is used for tailoring software systems to the customers' needs and to the available hardware. Runtime reconfigurable systems require special attention during the design phase. Especially during evolution one must think about not violating the reconfiguration requirements of the software. Generally, how the software reconfigures itself has to be modeled explicitly in the architectural model. Usually, the verification of the reconfiguration requirements is realized at the implementation phase increasing the development time. We address these problems with a novel process and a tool set for automating the verification of UML models with respect to runtime reconfiguration requirements. In this process, the UML models are converted into a graph-based model. The execution semantics of UML are modeled by graph transformation rules. Using these graph transformation rules and a graph production tool, the execution of the UML models is simulated. The simulation generates a state-space showing all possible reconfigurations of the models. The runtime reconfiguration requirements are expressed by computational tree logic (CTL) or with a visual state-based language (VSL), which is converted into CTL. The state-space is traversed with a verification algorithm for finding the states that satisfy the CTL formula. We also developed two mechanisms to provide error reports when the verification fails: 1) based on tracing the CTL formula to find the location where the formula evaluates to false. 2) based on a control automaton the execution sequence of the reconfiguration is provided (using VSL) and the simulation tries to generate this execution sequence. We conducted

experiments/case studies to evaluate the effectiveness of both mechanisms.

The second process and tool set are developed for computer aided detection of static program constraint violations. Software artifacts usually have static program constraints and these constraints should be satisfied in each reuse. In addition to this, the developers are also required to satisfy the coding conventions used by their organization. Because in a complex software system there are too many coding conventions and program constraints to be satisfied, it becomes a cumbersome task to check them all manually. Current tools which have been developed to computerize the program constraint violation checking use code querying and/or extensions to type systems. A limitation of these tools is that they work on abstract-syntax trees (ASTs) and do not provide adequate feedback when constraint violation is detected. The AST is at a different level of abstraction than the source code the developer works on, so constraints on program elements visible on the source code that the developers use and are familiar with cannot be verified. We developed a modeling language called Source Code Modeling Language (SCML) in which program elements from the source code can be represented. In the proposed process for constraint violation detection, the source code is converted into SCML models. The constraint detection is realized by graph transformation rules which are also modeled in SCML; the rules detect the violation and extract information from the SCML model of the source code to provide feedback on the location of the problem. This information can be queried from a querying mechanism that automatically searches the graph. The process has been applied to an industrial software system and to an open source software system.

The third process and tool set provide computer aided verification whether a design idiom can be used to implement a change request. The developers tend to implement evolution requests using software structures that are familiar to them; we call these structures design idioms. Implementing the design idioms requires the developer to follow a work-flow, which is a step-by-step description to implementing the design idiom. Each step of this work-flow has invariants that are crucial to the correct operation of the idiom and should be implemented. These invariants, however, have constraints that need to be satisfied before they are implemented. Usually, the applicability of a design idiom to a change request is tested manually. In our process, the work-flow for a change idiom is defined, and the invariants of each step are extracted from the existing source code by experts. Because the invariants are extracted from the source code, they may depend on program elements that are only visible at the source code. In addition, these invariants may include such program elements. The SCML meta-model includes these elements, so in this process the source code is converted to models in SCML. The verification of invariants are done over these models. Graph transformations are used for detecting whether the constraints of the invariants are satisfied or not. If the constraints are satisfied, then

the transformation rules add the code which implement the invariants. For a given design idiom and given source files, the work-flow is simulated. If all the steps of the work-flow can be executed, then all invariants of the design idioms are implemented in the resulting SCML model. Thus, the models are converted back to source file. If, on the other hand, a step cannot be executed, then the design idiom cannot be applied and the information about the failing step is presented. The approach has been applied to one open source and one industrial software system to show its applicability.

Contents

Chapter 1

Introduction

In order to keep competing on the market, to meet users' demands and to adapt to changes in the environment, software systems have to evolve [90]. Research on various industrial software systems has shown that a lot of time is spent on evolving and maintaining the software [50]. Because of this, in recent years a substantial amount of research has been performed that addresses various aspects of the software evolution problem.

This thesis presents research on software evolution conducted under the roof of the Darwin project [5]. The aim of this project is to understand and to provide solutions for the software evolution problems for very complex industrial software systems. Our research question in the Darwin project is to verify that an evolution of the software does not violate its invariants. We studied this research question both for runtime and compile-time evolution on two software components provided by our industrial partner. In the literature, tools and methods have been proposed that solve similar evolution problems; however, we identified certain drawbacks of these methods. To address these drawbacks, we developed three processes and supporting tools that enable computer-aided verification.

In the remainder of the present chapter we introduce these processes and tools. In the next section the problems which this thesis addresses and the drawbacks of the approaches in the literature that address similar problems are explained. Our solution approach is described and the contributions of the thesis are introduced in Section 1.2. In Section 1.3, we present an overview of the research on software evolution and where the processes proposed in thesis fit into this field of research. Finally, section 1.4 provides an overview on the organization of the thesis.

1.1 Problems which are Addressed in this Thesis

This thesis addresses 3 problems that we identified during our meetings with the architects/designers and observations of the developers while they were evolving the software. Our industrial partner in the Darwin project is an MRI (Magnetic Resonance Imaging) machine vendor. An MRI system consists of many hardware components that are controlled by the software; moreover, the captured images are converted to human understandable format by the software. Due to improvements in the imaging algorithms, hospital environments and hardware components, the requirements of the software evolve rapidly. These evolutions in the requirements should be implemented quickly, so the customers can quickly benefit from the improvements. The MRI software of our industrial partner is a complex multi-language, multi-paradigm, multi-component software: it consists of 31949 source files written in C, C++, C#, Perl, and in-house developed languages and has been evolving for years.

The research question behind the identified problems is to verify whether the software can evolve in the desired way without violating its invariants. In the literature, approaches that address this problem have been proposed; however, these approaches either were not able to work at the desired level of abstraction or they did not provide adequate guidelines on errors when the verification fails. This led to the development of 3 processes, detailed in the remaining chapters of this thesis. Below an overview of the problems this thesis addresses is presented:

1. **Verification of the Reconfiguration Requirements at the Implementation**:

   As discussed before, an MRI machine consists of different hardware components controlled by the software. A hardware component in an MRI machine model can be replaced by compatible versions of the same component. Thus, when a hardware component changes, the software has to reconfigure itself to use the new component. Besides hardware component changes, some software components need to be designed to be extensible; so the customer can purchase the set of extensions she/he desires. These extensions can also be added to the software without reinstallation; the software has to recognize the extensions and to reconfigure itself to use them.

   Software reconfiguration at runtime is achieved by changing the communication links between software components. Reconfiguration mechanisms are programming methods that allow such changes at runtime. The decisions on which reconfiguration mechanisms to use on the software are usually taken during the design phase. The selected mechanisms are specified in the de-

sign models, so the developers implement these mechanisms. Whether the selected reconfiguration mechanism reconfigures the software in the desired way, is commonly verified at the implementation phase. However, it may be too costly to change design decisions at the implementation phase.

In the literature, approaches that allow specification/verification of reconfiguration on the software component models are proposed [107, 88, 13]. Recently, the use of product line models is promoted in specifying reconfiguration [66]. The models used by these approaches are at a very high level of abstraction and they do not include the runtime behavior of the software system. As a consequence, the selected reconfiguration mechanisms and the point where these mechanisms are executed cannot be expressed with these models, making it hard to reason about runtime reconfiguration at the model level.

2. **Program Constraint Verification while Evolving the Software**:

   Usually, the software is evolved by reusing parts of it. One has to specify the constraints on the reused parts because the violations of these constraints may introduce errors to the software, hampering the benefits of reuse. However, a complex software system has many constraints and, during reuse, it becomes a cumbersome task to verify manually that the constraints are not violated.

   In the literature, tools based on predicate logic are used for checking constraints [65, 39]. We identified the following drawbacks which hampered the applicability of the proposed approaches in complex software systems:

   - These tools work on the abstract syntax tree (AST); however, we observed constraints that refer to program elements such as macros and comments that do not exist in the AST.
   - The error reports of these approached include information from the AST, which makes it hard to locate the violation of the constraint in the source code.

3. **Manual Testing on the Applicability of a Design Idiom**:

   We observed that the developers/designers tend to use software structures that they are familiar with in implementing the change requests; these structures are termed *design idioms* in the literature [116]. In the usage of design idioms, we identified two problems: 1) there is no way to test whether a design idiom can be used for a change request without trying to implement the change request with the idiom 2) the idioms are not well documented; as a result, it is hard for developers other than the experts on the idioms to use the idiom correctly.

   The main difference between the design idiom verification and other related approaches in the literature such as automated program transformations [122]

and refactoring transformations [102] is that in our approach we use an intermediate model rather than applying the transformations directly on the abstract syntax tree. The abstract syntax tree is at a different level abstraction than the source code seen by the developers. However, a design idiom may require the addition of other program elements like comments (due to the programming conventions followed by the industry) and macros. The meta-model we propose is a representation of the source code seen by the developers. As a result, the design idioms can be modeled to add program elements that do not exist in the AST.

1.2 Solution approach: Change Operators and Evolution Simulation

In the previous section, we have described the evolution problems this thesis addresses. The meta-problem of all these problems is to evaluate whether the software can be evolved in the desired way without violating the invariants. Usually, such an evaluation is done by trying to implement the design idiom or by executing runtime tests for runtime reconfiguration and design constraints. However, for all these problems the state of the software before the change and the state in which the software has to be after the change is known. It is possible to derive how the change happens (or the semantics of the change) by looking at the differences between these states. In the evolution simulation approach, the *change operators* capture the semantics of the change. These change operators can be modeled and they can be applied on the software to evaluate whether the software can be evolved using these operators. Thus, the evaluation can be done without any implementation or runtime tests.

A change operator has preconditions, software entities that the software should contain, and an algorithm that defines how the change is executed. A change operator is applied as follows: if the preconditions are satisfied on the software then the change operator's algorithm is executed. This is similar to the way graph transformations work. Because of this similarity and the availability of mature graph transformation tools, we model the change operators as graph transformations using a meta-model for representing the software as a graph. Thus, the simulation is done by a graph transformation tool that automatically applies the change operators. In our processes, we defined meta-models for representing both design level models (e.g. UML models [55]) and source code (focused on Java and C/C++ source code but can be extended to represent other languages). We also developed tools to convert UML models and source code written in Java and C to graphs using the developed meta-models.

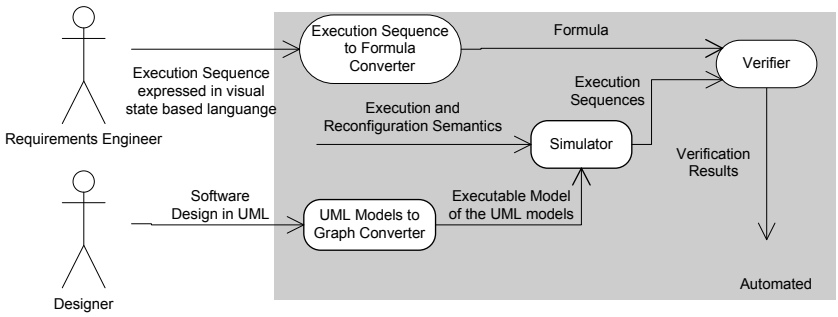


Figure 1.1: The process for verifying the reconfiguration requirements of UML models. The ellipses represent the tools and the arrows are the inputs to the tools.

We specialized the evolution simulation approach and developed tools to support these specializations to address the evolution problems described in the previous section. Thus, the contributions of this thesis are tools and processes that provide computer-aided verification of reconfiguration requirements, design constraints and the applicability of change idioms. In this section, we briefly introduce these processes.

1.2.1 Run-time Reconfiguration Verification of UML models

A runtime reconfiguration is an anticipated change on the executing software caused by variants in the environment. Runtime reconfiguration is achieved by changing the connections between different software modules [107]. We call the software structures that allow such changes *reconfiguration mechanisms*. Usually, these mechanisms are specified in the design models; however, there is no way to evaluate whether the software can reach the desired configuration with the specified mechanisms on these models.

To apply the evolution simulation approach, we modeled change operators that capture the semantics of the reconfiguration mechanisms. This is not sufficient to reason about the reconfiguration, because the reconfiguration mechanisms are executed when the execution of the software reaches certain points. As a result, the simulation should simulate the execution of the software with the reconfiguration mechanisms. We modeled graph transformation rules that capture the execution semantics of object-oriented software; however, because the approach is applied on design models, these transformation rules only cover the semantics that is included in the design model.

Figure 1.1 presents the process for verifying the reconfiguration requirements of the UML models. The requirements engineer specifies an execution sequence the software has to support or an execution invariant. This specification is done using a visual state-based language (VSL). We developed a converter tool that converts the reconfiguration specifications in VSL into CTL formulas. The designer models the design of the software in UML. Another converter tool converts these models into a graph-based model. With the change operators and the transformation rules modeling the execution semantics of the software, the simulator generates a state-space showing all possible execution sequences; here, the simulation yields more than one execution sequence due to reconfiguration. An evaluation algorithm evaluates the formula by searching the generated state-space to find whether the software design models support the specified execution sequence or the execution invariant. In this way, the designer is able to verify the reconfigurability of the software at the design level without any implementation.

1.2.2 Verification of Program Constraints

Constraints should be satisfied in order to effectively reuse a program. Because a complex program has many constraints, manually checking these constraints during reuse hampers the benefits of the reuse. We specialized the evolution simulation approach for providing computer-aided program constraint verification. In this specialization, the preconditions of the change operators model the constraints. These preconditions of change operator are used for detecting whether the constraint they model is violated or not. The algorithm of these operators, on the other hand, does not change the software, it is used to extract information from the software about the location of the constraint violation. Compared to other approaches from the literature, with our approach the constraint checking is realized at the source code over program elements visible to the developer. We employ a meta-model that covers the source code level program elements from structural and object-oriented languages.

A software system may contain too many constraints, so it may be hard to manage the change operators. The design constraint verification processes make use of a repository. We developed a repository manager tool which allows the developer to place constraints in the repository, search for the constraints and see the description of the constraints. It is important to note that the change operators in the repository are stored as templates. That is, they do not contain names of the software entities. This is done to increase the reuse of the change operators: the structure of the software may be an invariant but the names of the software entities in the structure may change. The developer checks-out the constraints she/he wants to verify from

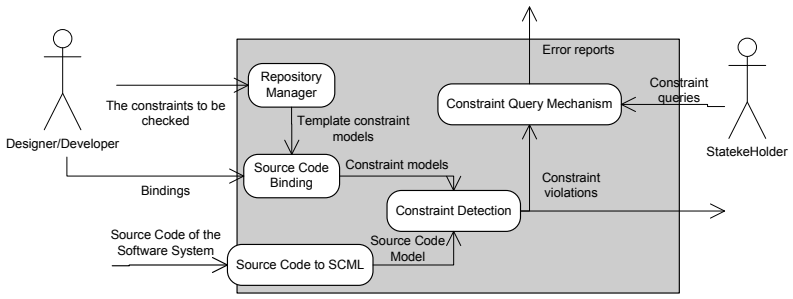


Figure 1.2: The process for verifying of the program constraints. The ellipses represent the tools and the arrows are the inputs of the tools.

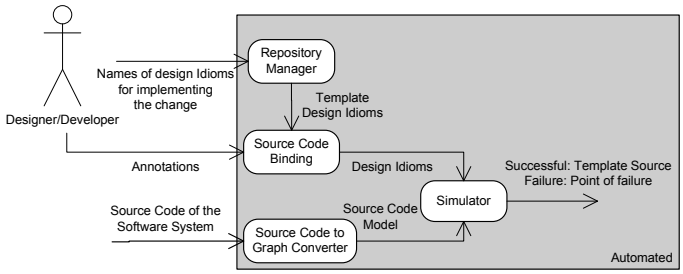


Figure 1.3: The process for verifying of the Design Idioms. The ellipses represent the tools and the arrows are the inputs of the tools.

the repository. During this check-out procedure, if the change operator requires the names of the software entities to be supplied, the repository manager asks the developer to supply these names. Figure 1.2 presents an overview of how this process works.

1.2.3 Verification of Design Idioms at the Source Code

The verification of whether a design idiom can be used for implementation of a change request is done by simulating the implementation of the idiom. Here, the change operators are the steps of the design idiom, where the preconditions include the software entities required to implement that step and the algorithm is the implementation of the step. We chose to model each step as a change operator rather than the complete idiom because we identified that certain idioms have steps that depend on certain conditions. Thus, besides the change operators the simulation of the implementation of the design idiom requires the work-flow the developer follows in implementing the design idiom. We used a control automaton to model the work-flows of the design idioms.

Figure 1.3 depicts an overview on how the verification process works with the parts that are automated by the tools. Here, the inputs of the simulator are the work-flow and the change operators of the idiom. The simulator follows the work-flow and tries to apply the change operators. If all the steps of the work-flow are completed then the change operator can be used. If, on the other hand, a step cannot be applied, then this means that either the invariants of the idiom or the invariants of the software are violated when the design idiom is used.

In this approach, we again use meta-modeling and work at the source code with the program elements visible to the developer. The reason for this is twofold: 1) the design idioms have invariants over program elements visible at the source code (e.g. macros) 2) the generated template source should be familiar to the developers.

We again use a repository to manage the design idioms. The change operators in the repository are stored as templates and require to be bound to the software. This binding is done by providing annotations in the form *Class_Converter = OperFrameConverter*. Here, the left hand-side of the assignment is the template name of the software entity the change operator is going to work on and the right hand-side of the assignment is the actual name used in the software.

1.3 An Overview of the Literature on Software Evolution

The term software evolution first appeared in the software engineering literature in 1970s by a study conducted by Lehmann et al. [20]. In this study the authors have measured the complexity, size, cost and maintenance of 20 releases of the OS/360 operating system, based on its source code. All 20 versions of this software system have shown an increasing trend in all measures. This study showed that evolving software systems is a very costly operation. The demands of the market, however, do not allow much time to be spent on implementing the changes to the software caused by evolution.

Following this observation, detailed analysis on evolving software systems has identified the following problems that cause evolving software to be costly:

- Wrong predications about the evolution procedure.
- Incorrect assumptions on the parts of the software effected by the change.
- The software is not designed with evolution in mind.

- While implementing the change, developers introduce new errors in the software.
- Lack of processes for verifying whether the software supports the desired evolution.

The research on software evolution has studied these problems and produced the following solutions to ease the software evolution process and to reduce the cost/effort spent in evolving the software: anticipation of changes, evolution mechanisms and software evolution verification. Below, we present an overview on the literature about software evolution categorized according to these solutions (the reader is referred to the literature [26, 97] for more detailed surveys):

- **Anticipation of Changes**: The principle of change anticipation is providing predictions about future changes by analyzing the software. The research on providing predictions about software evolution can be further categorized into two sub-categories according to the sources of information they use.

  The first category uses the history of the software to provide predictions on the future changes. Version control systems like CVS [4] are widely adopted tools for keeping track of changes made to the software. The initial version of software system is *committed* to a repository supplied by the version control system. To make a change, a developer first *checks-out* the source files that need to be changed. The developer *commits* these source files after making the changes. For each commit, the version control system records the changes made, the developer that made the change and an optional description of the change (entered by the developer). Thus, these records hold the whole evolution history of the software, and this category research derives statistics that can be used to predict future changes based on this history. Initial studies used the history analysis to understand the evolution process and predict the growth of the software so that organizations can better adapt their processes and budgets [89]. The principle measure used in these studies is the number of components. Kemerer and Slaughter [82] have shown that using different metrics can result in different predictions about the growth of the software. In this study, they conducted time, sequence and gamma analysis on two different software systems. An important observation of this analysis is that these software systems start their evolution cycle with similar activities, like addition of new modules.

  Later studies focused on providing predictions on possibly problematic locations in the software; for example, Graves et al. [61] use the history of bug-fixes to find the most probable locations to contain bugs when the software evolves.

Usually, a change requires more than one part in the software to change; however, the developer may forget to identify all the parts related to the change, which in turn introduces errors to the software. Zimmermann et al. [134] analyze the change history to find the lines of code that change together to address this problem. During evolution, the developer is presented with the lines of code that change most frequently together with the lines the developer is changing. The aim is to prevent the developer from forgetting to change a related location when she/he evolves the software.

The second category of research uses change impact analysis for change anticipation. The aim of change impact analysis is identifying all the parts of the software that are effected by the change [17]. A program slice is a part of a program that has an effect on a computed value; a slicing technique allows the developers to identify related slices [119]. Due to similarity between program slicing and change impact analysis, many slicing techniques can be used in the context of change impact analysis. Recently, change impact analysis that focuses on the semantics of changes in the object-oriented software is proposed [114]. These approaches are extended to include the semantics of changes on aspect oriented software [132]. In this way, the effects of the change on both object-oriented and aspect-oriented software can be assessed.

- **Evolution Mechanisms**: The research in this category has built tools, processes and software structures that can ease/enable the evolution of the software; we call them evolution mechanisms. There are two groups of evolution mechanisms: built-in mechanisms and computer-aided implementation mechanisms. The built-in mechanisms address the problem of designing the software with evolution in mind; these mechanisms should be implemented in the software, so that they can be used in the future to evolve the software. The computer-aided implementation mechanisms, on the other hand, are not required to be included in the software. These mechanisms try to address the problem of developers introducing errors to the software during change implementation by raising the level of abstraction. These mechanisms transform the software to an evolved state using predefined transformations. The developer describes how the change is going to be implemented in terms of these transformations and the tools provided with these mechanisms make the modifications to the software. Below we provide example evolution mechanisms from the literature for these sub-categories:

  1. **Built-in Evolution Mechanisms**: If future changes to a software component are anticipated, then design patterns [59] that ease the implementation of the anticipated changes can be used in this component. For example, the visitor design pattern can be implemented for a class for

which it is anticipated that more methods will be added. It is important to know which evolution problems can be eased with which design patterns; using the wrong design pattern to an evolution problem can increase the effort spent in evolving the software. To address this problem, in the literature approaches for understanding the relation between evolution problems and design patterns are developed [98, 31].

The execution of some software systems (like telephone line managers) cannot be aborted to evolve the software. These software systems need to be built with evolution mechanisms that allow changes to be incorporated into the software while it is running. In the literature much attention is given to developing evolution mechanisms that allow changes to the software to be incorporated without stopping the execution of the software; examples of such mechanisms include the building of address transition tables to replace functions [58], relinking the program at runtime [67], modifying the Java virtual machine to support type-safe dynamic replacement of the loaded (and executing) classes [94, 83] and using aspect-oriented programming [52].

Runtime reconfiguration allows the software systems to be adapted to the users' desires and to the environment. Usually, such software systems are designed with configurable connections between components [107]: the software is shipped with core and adapted components and an initial configuration that connects the core components to a selected adapted component. This connection is changed to the desired adapted component to reconfigure the component. The most widely used built-in evolution mechanisms that allow the connections between the components to change are polymorphism and reflection [81, 28, 45, 130].

2. **Computer-aided Evolution Mechanisms**: This line of research introduces methods and tools that automate the implementation of changes. The research on automated implementation of changes can be further divided into three groups according to the amount of source code generation related to the change as follows:

   (a) *Full Source Code Generation of the Change*: In the literature, program transformations are proposed as a method for automating modifications of programs and their application as evolution mechanisms has been studied [122]. The building blocks of program transformations are transformation rules that are applied to a fragment of a program: the transformation rule detects a pattern and replaces the detected pattern with the pattern defined in the rule. The transformation rules are combined with *programmable strategies* that place an application order on the transformation rules. In this way, the change

implementation is moved to a higher level of abstraction; rather than modifying the source code (and possibly introducing syntactical errors) the developer programs the change in terms of the transformation rules. The tools then apply the transformation rules and, if all transformation rules are applied successfully, the source code implementing the whole of the change is generated. In the literature, many tools are developed that allow transformations in the abstract syntax-tree (AST) [18, 115, 121, 120]. These tools are extended with pretty-printers so that the transformed AST can be converted back to source code format.

(b) *Template Source Code Generation of the Change*: With these evolution mechanisms, some part of the implementation for the change is generated by the tools and the rest is implemented by the developers. As discussed before, design patterns need to be built-in to the software so that they can be used to evolve the software in the future. However, their benefits on easing the evolution may be hampered if the design pattern is not documented or if the design pattern is not evolved correctly. In the literature, approaches for automated design pattern recognition and evolution are proposed [32, 133, 105, 101, 29]. Once the design pattern is detected by these approaches, they generate the structure that obeys the constraints of the design pattern. The developer, then, implements the change on this structure.

(c) *No Source Code Generation Related to the Change*: The proposed mechanisms in this category are divided into two groups. The first group of mechanisms is used for improving the structure of the software such that the implementation of future changes is eased. In the literature these processes are called *refactoring transformations*. Refactoring transformations are special program transformations that aim to improve the structure of the program [106]. The main difference between program transformations and refactoring transformations is that refactoring transformations do not alter the external behavior of the program. A refactoring transformation is specified as a set a preconditions, invariants and an algorithm: the preconditions are program elements that are required in order to apply the refactoring transformation, the invariants are program elements that should be left untouched by the transformation rule and the algorithm specifies how the refactoring happens. Due to the benefits of refactorings, a substantial number of tools are developed for automating refactoring transformations for Java; examples include [78, 118, 105, 117]. The presence of C preprocessor statements makes it hard to develop refactoring tools for C++. To overcome this problem, ap-

proaches that map preprocessed code to actual source code are developed [124]. Today, popular software development environments such as Eclipse [6] and Visual Studio [12] support semi-automated refactoring transformations.

Refactoring a framework may cause changes to interfaces of the framework. This in turn causes backward compatibility problems where the software developed with the old version of the framework needs to adapted to the new interface. Şavga et al. [40] propose *comeback* transformations to generate compatibility layers between the refactored version of the framework and software that uses the old versions of the framework. A comeback transformation is an inverse of a refactoring transformation. Comback transformations are not applied to the types of the framework directly; an adapter for the type is generated and to these adapters the comeback transformations are applied. In this way, the software is able to use the refactored framework without any modifications to either the framework or the software.

The mechanisms belonging to the second group are used for verifying whether the developers have satisfied the constraints of a program while evolving the software. A complex program may have too many constraints [108] and the developer needs to satisfy these constraints in order to correctly reuse it. However, due to poor documentation and complexity of the software systems, it is a cumbersome task to verify whether the implemented change satisfies the constraints of the software. In the literature, approaches are proposed that use predicate logic for computer-aided static constraint verification [39, 47, 95, 36, 65, 51]. In all these approaches, the programs elements (the structure or the AST) are converted to predicates in a Prolog-like language. The constraints are expressed as rules over the predicates of the program elements. A logic engine evaluates these rules and outputs the rules with predicates that evaluate to false; these are the constraints that are violated. Other approaches to static constraint checking use languages that combine first-order logic with a term language [70] and extensions to type systems [23].

- **Software Evolution Verification**: This line of research aims at verifying whether the software can evolve in the desired way. The verification effort, here, is focused on the higher-level models of the software, because it may be too costly to fix errors at the implementation. Scenario-based analysis is the most intuitive method for verifying software architectures with respect to their requirements [44]. To verify how well the software systems handle

an evolution requirement, scenarios of various evolutions are specified by the stakeholders. Then, the designer shows how these scenarios are handled by the software architecture. This procedure ends when the addition of a new scenario is handled as expected by the architecture [80, 93]. Scenario-based analysis is specialized to support many aspects of software evolution: examples include finding parts of an architecture that are hard to modify [21], comparing evolution with other quality attributes [79] and verifying runtime evolution requirements [91].

In all scenario analysis methods, the verification is done manually by the designer. Because of this, the evaluation process may be imprecise and labor intensive. Especially for runtime evolution requirements, the designer may miss certain runtime properties of the software architecture, causing wrong conclusions to be drawn about the software architecture. To address these problems, formal semantics for software architecture evolution are defined and the evolution of the architecture is simulated [24]. For example, the runtime evolution of the components of the architecture is specified as temporal logic formulas [13]. Then, the correctness of the evolution of the architecture is verified by using a theorem prover. Due to the similarity between runtime reconfiguration and product lines, approaches that use product variability models to model the configurable components of the software architecture are proposed [130, 66]. These approaches provide verification on these models of whether a desired configuration can be reached or not.

Above we presented three solutions provided in the literature to ease the software evolution process. Software evolution is a very complex problem effecting software artifacts from higher-level models to the source code of the program. Due to this, the research on the evolution problem takes a sub-problem and develops tools and methods to address this sub-problem. The sub-problems of these approaches differ but the common aim is to ease the software evolution process with the proposed tools and methods.

The process used for verification of the reconfiguration requirements and the process for verification on the applicability of a design idiom belong to the software evolution verification category of the solutions produced by the software evolution research. Although the main aim is not code generation, the process on the verification of the usability of design idioms is used for template code generation. So, the third approach also belongs to the category of computer-aided evolution mechanisms. The process for verification of the program constraints also belongs to this category.

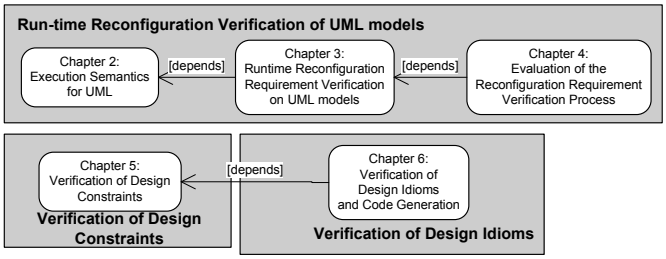


Figure 1.4: The distribution of the evolution problems addressed by this thesis over the chapters.

1.4 Overview of the Thesis

Figure 1.4 depicts the distribution of the problems described in the previous section over the chapters of this thesis. Here, the arrows between the chapters (the ellipses) show the dependency relation between the chapters; for example, chapter 3 depends on the content of chapter 2. Below the contents of the chapters are detailed:

Chapter 2 introduces the meta-model of the *design configuration modeling language* (DCML); this language is used for creating *design configuration models* which are graph-based representation of UML models. The graph transformations rules, which are also modeled in DCML, representing the execution semantics for UML models are also detailed in this chapter.

Chapter 3 describes the process for computer aided verification of runtime reconfiguration requirements on UML models. Here, the reconfiguration requirements can be specified using Computational Tree Logic (CTL) formulas or using the visual state-based language we developed called *VSL*. To specify the reconfiguration mechanisms on the UML models, extensions to UML are provided. The semantics of these mechanisms are also modeled using graph transformation rules. Finally, the chapter describes two feedback mechanisms that provide guidelines to the designers on the possible location of the problem when the verification of a reconfiguration requirement fails.

Chapter 4 presents the evaluation of the reconfiguration requirement verification process. First, a case study conducted with a designer from the industry is presented. The aim of this case study is to compare the outcome of the verification on the design models with the implementation of the same design. In this case study, the UML models of an industrial software are simulated and the reconfiguration requirements of this software are verified. The results of the verification are compared with manual evaluation on the implementation of the tool.

Using the UML models of this industrial software, two experiments were conducted with computer science master students to test the effectiveness of the feedback mechanisms. In these experiments the students were divided into two groups: one group used tools implementing the feedback mechanism and the other group used manual evaluation by tracing the UML models. The students were asked to evaluate reconfiguration requirements and to correct the UML models if the evaluation fails. Statistical analysis is used to show that there is a significant difference between the number of errors made by the students who manually evaluated the requirements and the number of errors made by the students who worked with the tool implementing the feedback mechanism.

Chapter 5 details the process and tools used for computer aided program constraint verification. We introduce the Source Code Modeling Language (SCML); this modeling language is used for representing source code as seen by the developers. The approach has been applied to one open source and one industrial software system.

Chapter 6 details the process and the tools used for computer aided design idiom verification. This process is also applied to the different versions of one open source and one industrial software system.

Chapter 7 presents our conclusions.

Chapter 2

Defining Execution Semantics for UML

For expressing software systems, high-level models are increasingly used in practice. Verification of such models with respect to the requirements is important because it allows the stakeholders to capture requirement realization errors in the design level of the software life-cycle. Verification of requirements at earlier levels than the implementation level is beneficial because correcting design errors and/or design decisions at the implementation level can be too costly.

UML class diagrams provide an overall view of the structure of the software system. The sample execution scenarios of the structure are depicted by UML sequence diagrams. A common practice in the industry is to manually trace these diagrams for requirements verification.

Manual tracing of the diagrams may lead to wrong conclusions. Moreover, certain requirements, like runtime reconfiguration, heavily rely on object-oriented composition mechanisms (e.g. polymorphism), putting more burden on manual tracing (e.g. requires tracing the sequence diagrams and the inheritance hierarchy).

To allow computer-aided verification of the requirements on these UML diagrams, formal execution semantics should be defined. We defined execution semantics of UML sequence diagrams through graph transformations. Using a graph-production tool the execution of the sequence diagrams can be simulated. This simulation generates a state-space on which various verification algorithms can be run. We use GROOVE [111] as the graph-production tool.

Our focus is on verifying reconfiguration requirements and because runtime reconfiguration relies heavily on modifying the composition of the software system (e.g.

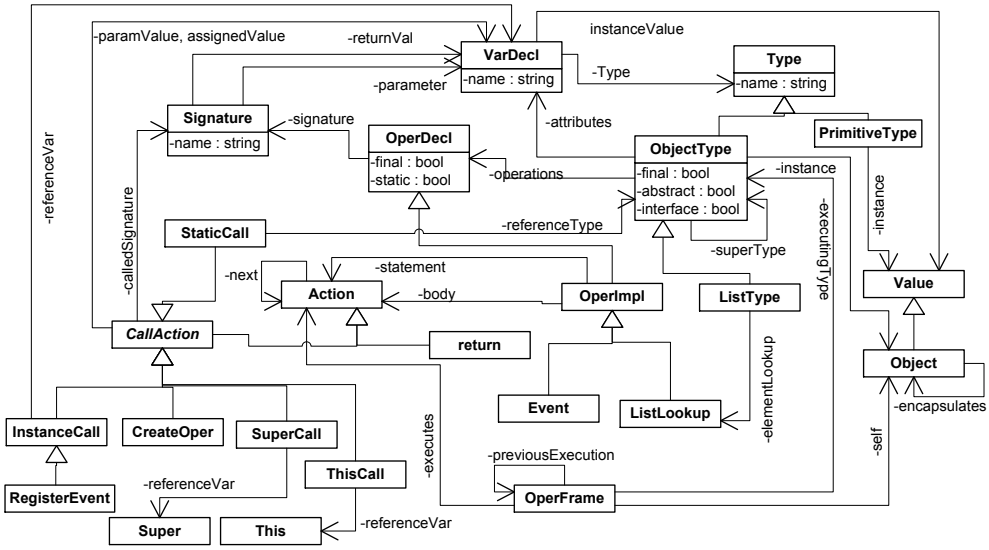
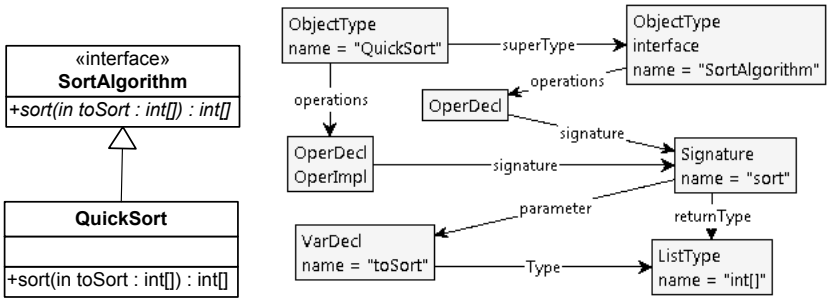


Figure 2.1: The meta-model of DCML.

by polymorphism), the execution semantics we provide is very close to actual execution of an Object-Oriented software. This chapter presents how we model UML class and sequence diagrams as graphs and describes the execution semantics.

2.1 The Design Configuration Modeling Language

The UML sequence diagrams depict the execution sequences in order to provide an overview of the interaction between objects in the software systems. Due to mechanisms such as conditional execution and polymorphism, the software system may support executions other than the ones depicted with the sequence diagrams. These hidden interactions may introduce bugs to the software when the sequence diagrams are implemented. In order to prevent the introduction of these bugs to the software system, there should be a way to reason about the executions supported by the diagrams. This reasoning requires the sequence diagrams to be simulated as close to the actual execution of an object-oriented (OO) software as possible. However, the sequence diagrams do not include model elements like execution frames that allow an OO like execution simulation. The Design Configuration Language (DCML) includes these elements and allows one to model an OO software runtime for UML sequence diagrams. In our approach, the DCML models (DCMs) are represented as graphs since the OO like execution semantics are defined as graph transformation rules. The DCMs are not full semantic representation of OO software, they only include elements that can be modeled with UML class and sequence diagrams. A



(a) (b)

Figure 2.2: a) The class diagram of the class *QuickSort* b) The DCM of the class *QuickSort*

DCM is generated from one class diagram and at least one sequence diagram.

Figure 2.1 depicts the meta-model of DCML. In the meta-model, the abbreviations var, oper, decl and impl stand for variable, operations, declaration, and implementation respectively.

The static structure of object-oriented software in UML models and in OO programs is similar; for example, classes have attributes, operations and super-classes. Because of this similarity, in our graph-based model the structure of the object-oriented systems is represented by similar graph elements (like *ObjectType)* as proposed by Kastenberg et al. [76]. The details of the dynamic structure, on the other hand, is different between OO programs and UML models; thus, the statements (e.g. call actions) and the elements that are used during simulation are modeled differently in DCML.

Classes and interfaces are represented by nodes labeled as *ObjectType*. The generalization/implementation between classes/interfaces are represented with edges labeled as *superType*. Figure 2.2-(a) presents the UML class diagram of the class *QuickSort* which implements the interface *SortAlgorithm*. Figure 2.2-(b) depicts the DCML equivalent of this class diagram. Here, the class *QuickSort* is represented by the object-type node labeled *QuickSort* and the object-type node *SortAlgorithm* represents the interface *SortAlgorithm*. These nodes are connected by the edge labeled *superType* to show that at runtime the object-type *SortAlgorithm* is a super-type of the object-type *QuickSort*.

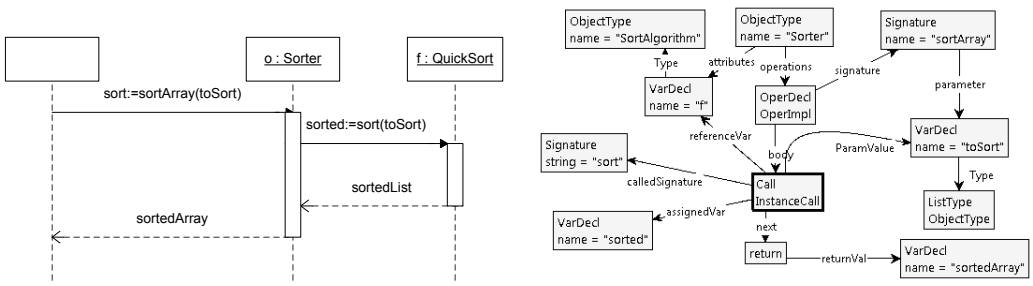
The attributes of classes are represented by nodes labeled as *VarDecl* (variable declaration nodes) that are connected to the object-type nodes with edges labeled *attributes*. The edge labeled *operations* connects an object-type to a method of that type. Abstract methods are represented by nodes labeled as *OperDecl* (op-

eration declaration nodes) and methods with implementation are represented by nodes labeled as both *OperImpl* (operation implementation nodes) and *OperDecl*. The interface *SortAlgorithm* in Figure 2.2-(a) has an abstract method; in DCML (Figure 2.2-(b)) this is shown by the edge labeled as *operations* connecting the object-type *SortAlgorithm* to an *OperDecl* node. The class *QuickSort*, on the other hand, has an implemented method; thus, in DCML the object-type *QuickSort* is connected to an *OperImpl* node with the an edge labeled as *operations*.

The DCML separates methods and the signatures of the methods. The main reason for the separation is to model method overriding at runtime. Each unique signature in the class diagram is converted to a node labeled as *Signature*. In the class diagram of Figure 2.2-(a), there is one unique signature named *sort* that takes an integer array and returns an integer array. As a result, in DCML there is only one signature node which represents this signature. The operation declaration node of the object-type *SortAlgorithm* and the operation implementation node of the object-type *QuickSort* are both connected to this signature node by an edge labeled *signature*. This shows that the object-type *SortAlgorithm* is declaring a method whose signature name is *sort* and the sub-type *QuickSort* is implementing this method. The parameters of signatures are represented by variable declaration nodes connected to signatures nodes by edges labeled as *parameter*. The return type of the signature is represented by connecting the signature node to a type node by an edge labeled *returnType*.

The implementations of the methods are extracted from sequence diagrams. The implementation of a method in DCML consists of *CallActions* and *ReturnActions*. The first action of the method is connected by an edge labeled *body* to the operation implementation node representing the method. The actions of a method are ordered by edges labeled *next*. Figure 2.3-(a) presents a sequence diagram with an instance of the class *Sorter* that has received the call *sortArray*. In the focus control of this call action, a call from this instance of the class *Sorter* is made and then the focus control ends with a return message. In DCML, this call and return message are put into the body of the method *sortArray* because these actions are made during the focus control of this method. In Figure 2.3-(b) the call action is the emphasized node. Here, this action is connected to the operation implementation node (representing the method *sortArray*) by the body edge because it is the first action.

The model supports *5* kinds of call actions: the calls to instances (*InstanceCall*), create actions (*CreateOper*), super method calls (*SuperCall*), self calls (*ThisCall*) and static method calls (*StaticCall*). The call to the class *QuickSort*'s sort method in the sequence diagram (Figure 2.3-(a)) is an instance call; it is a call action to an instance of the class *QuickSort* from an instance of the class *Sorter*. The instance that is going to receive the call is labeled *f*. DCML only supports communication between objects through encapsulation. As a result, the classifier names are represented as variables



(a) (b)

Figure 2.3: a) A sequence diagram showing an execution scenario of the class *QuickSort* b) The DCM of the same execution scenario

which hold the object that is going to receive the call. For this call action, the conversion tries to locate whether a variable declaration node with name *f* is present in the scope of the call (i.e. it is an attribute in the class *Sorter* or it is declared in the signature of the method *Sorter.sort()*). If it is found, then the edge labeled *referenceVar* is drawn from the call node to the variable declaration node; if it is not found, the conversion adds a variable declaration node to the method and adds the edge labeled *referenceVar*. In the example, there is an attribute named *f* so the edge labeled *referenceVar* is drawn from the call node (the emphasized node in Figure 2.3-(b) to this variable node. The signature that the call action calls is represented by connecting the call node to the signature node by an edge labeled *calledSignature*.

A call action node can be connected to variable declaration nodes by edges labeled *paramValue* to model the parameters the call passes. The parameters are converted from the arguments of the call action specified in the sequence diagram. Each argument is converted to a variable declaration node with the same name. In Figure 2.3-(a), the call action *sort* passes the argument *toSort*. In the DCM of this call action, Figure 2.3-(b), this is converted as a variable with the name *toSort* connected to the node representing this call action (the emphasized node).

In DCML, the variables that get assigned the return value of the method are modeled by a variable declaration node connected to the call action node by an edge labeled *assignedVar*. For example, the call action *sort* in Figure 2.3-(a) assigns the return value to a variable named *sorted*; in the DCML version of this call action *sorted* is a variable connected to the call action node (the emphasized node in Figure 2.3-(b)).

The values of the arguments are represented by nodes labeled as *Value* that are connected to the variable declaration node representing the argument. The value node is only converted if a name or a unique id is specified in the design (in UML, it is possible to give a name to a value similar to the name of an object). In Figure 2.3-

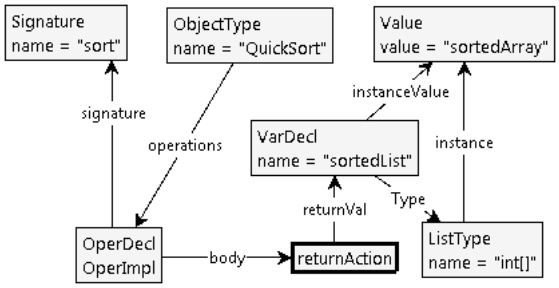


Figure 2.4: The DCM of the the method *QuickSort.sort()* detailing the return action.

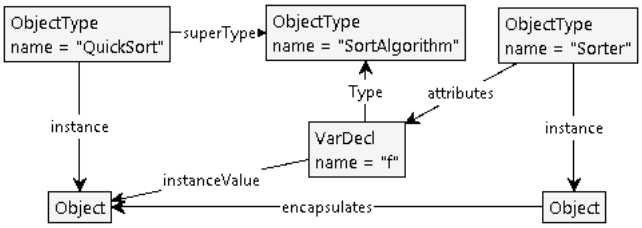


Figure 2.5: The instance of the class *Sorter* encapsulates an instance of the class *QuickSort*; the attribute *f* holds this instance.

(a), we see that the call action *sort* returns the message *sortedList*. Figure 2.4 depicts this in DCML. Here, the body of the method *QuickSort.sort* contains only the return action (the emphasized node), because in the sequence diagram no other action is specified in the focus control of this method. The returned message is represented by a variable declaration with the same name. Although not shown in the sequence diagram, the value for this argument is set and named *sortedArray*. In DCML, this value is represented by a value node whose value attribute is set to *sortedArray*. The variable *sortedList* holds this value; so, in DCML the variable *sortedList* is connected to the value *sortedArray* by an edge labeled *instanceValue*. The value is an instance of the list-type *int[]*; this is represented by an edge labeled *instance* connecting the list-type node to the value node.

In the sequence diagram, Figure 2.3-(a), *f* is an instance of the class *QuickSort*. In DCML, *f* is converted to a variable, which holds an instance of the class *QuickSort*. This is depicted in Figure 2.5. Here, the instance of the class *Sorter* is connected to the instance of the class *QuickSort* by an edge labeled as *encapsulates* (encapsulates edge); that is, in the scope of this instance of the class *Sorter* the variable *f* holds an instance of the class *QuickSort*. Because a DCM can be generated from more than one sequence diagram, a variable can have more then one instance value. During simulation, the values of the variables at the executing frame are resolved with the encapsulates edges.

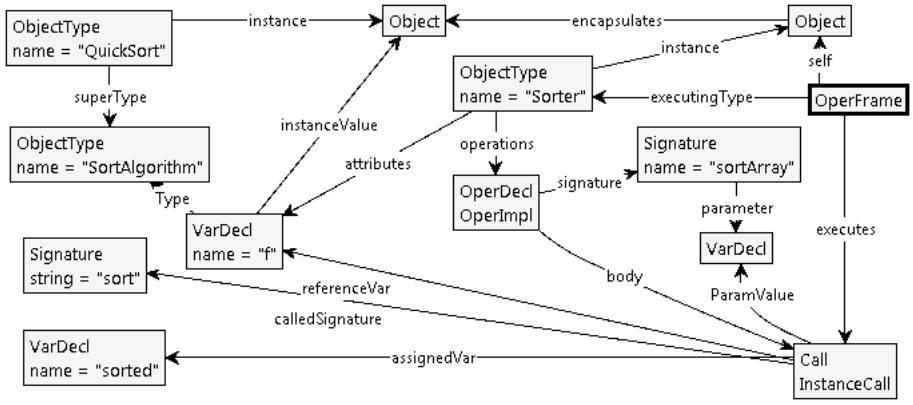


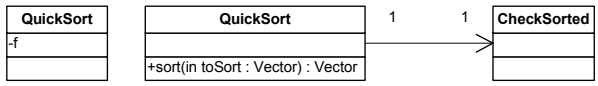
Figure 2.6: The snapshot of the operation frame executing the call *f.sort()*.

DCML contains the notion of an operation frame, modeled by nodes labeled as *OperFrame*. With such nodes, during simulation, the object that is currently executing, the type that contains the called method and the statement that is being executed are marked. Figure 2.6 shows a snapshot from the simulation of the sequence diagram of Figure 2.3-(a). The operation frame node is the emphasized node. This node is connected to an instance of the class *Sorter* by an edge labeled *self*; thus, this object is the object that is currently executing. Following the encapsulates edge, it is possible to resolve the value of the attribute *f* as an instance of the class *QuickSort*. The edge labeled as *executes* connecting the frame node to an action, marks the action the simulation is currently executing; for this snapshot it is an instance call. The conversion algorithm adds an operation frame node which marks the first action of the sequence diagram as the action that is being executed. Thus, the simulation starts executing from that action.

2.1.1 Conversion from UML to DCML

The open source UML editor ArgoUML [2] supports import and export of sequence diagrams in XMI. Using the XMI interface of ArgoUML, we have implemented a translator to convert UML models to DCMs. The translator executes in two steps, class diagram conversion and sequence diagram(s) conversion. The conversion requires one class diagram and at least one sequence diagram. When more than one sequence diagrams are presented, the conversion algorithm marks the first call in the first sequence diagram as the statement the simulation starts from.

The conversion from UML-to-DCML places certain restrictions on the UML models because DCML only supports interaction between objects through encapsulation.



(a) (b)

Figure 2.7: UML class diagrams with errors: a) the type of attribute is not defined b) the *end* name for the association is not defined

In Appendix A conversions of UML elements are detailed. Below, we briefly describe the constraints one needs to comply to generate UML models that can be converted to DCML models:

- The types of attributes and parameters should be specified. For example, the class shown in Figure 2.7-(a) cannot be converted to DCML because the type of the attribute *f* is not specified.

- The *end* names for associations should be specified. DCML only supports communication between object through encapsulation and, because of this, every association (including special associations like composition and aggregation) is converted into an attribute. Figure 2.7-(b) shows an association that cannot be converted to DCML because of the missing end name.

- The arguments of a call action should be specified and cannot refer to values. In UML, the arguments of a call action model the variables the method passes to another method through the call and they are specified with two fields: the *name* of the argument and an optional *value* where one can specify a name for the value the argument holds. It is common practice to specify a value as the name of the argument which, in most UML editors, displays a constant value as an argument. However, the DCML does not support values as arguments to calls, as shown in the DCML meta-model (Figure 2.1). The converter converts every argument of a call action to variable declaration nodes whose names are the same as the names of the arguments. If a value is specified instead of a name in the name field of an argument, then the converter generates a variable declaration node whose name is the constant value causing errors during simulation. Figure 2.8 demonstrates the error caused when a constant value is specified as the name of an argument. Here, Figure 2.8-(a) and (b) present the UML models of a software system with a call action that has four arguments and Figure 2.8-(c) shows the DCML model generated from these diagrams. The first argument named *toSort* is a parameter of the method *Sort* as specified in the class *Quicksort* (Figure 2.8-(a)); because a variable declaration with the name *toSort* already exists in the scope of the method *Sort*, the converter only adds the edge labeled *paramValue* from the call action

node $n16$ to the node $n15$ representing this variable. Similarly, the second argument *f* shares the same name as the attribute *f* of the class *QuickSort* and, for this argument, the converter adds the edge labeled *paramValue* from the call action node (node $n16$) to the variable declaration node representing the attribute *f* (node $n2$). A variable with the name *sortOptions* is not previously declared in the scope of the method *Sort* (i.e. it is neither a parameter of the method *Sort* nor an attribute of the class *QuickSort*); thus, the convert algorithm adds a variable declaration node for this argument. This variable declaration node is the node $n12$ in Figure 2.8-(c). The fourth parameter is the value 0. The designer specified this argument with name 0 and without a value. Because DCML does not support constant values as arguments, the converter converts the constant to a variable 0 and no value is converted because none is specified as shown in node $n21$. This causes an error during simulation as the execution semantics for parameter passing will not be able to resolve the value for the variable 0. This error can be circumvented by specifying a name of the argument and specifying the value 0 at the field value of the argument.

- All instance values should be specified as classifiers, even if the classifier does not receive call. The classifiers are represented by object/value nodes in DCML; the variable that holds the object node is resolved from the name of the classifier and the type whose instance the object/value node represents is resolved from the type of the classifier. Because in a sequence diagram values/objects are passed between objects, it is important to show from which object a passed value/object initiated. Also, it is important to show which instances of classes play a role in the sequence diagram. It is common practice, however, to omit classifiers for values of primitive types and for objects that do not receive a call. For example, in Figure 2.8-(b) the classifier named *f* shows an object that is an instance of the class *SortAlgorithm*. One could only assume that the attribute *f* holds an object that is type-compatible with the class *SortAlgorithm* without this classifier. This makes it hard to grasp which objects play a role in the interactions modeled in a sequence diagram. To solve this problem, we enforce all classifiers to be explicit in the model, as shown in Figure 2.8-(b). If the converter cannot resolve the classifier for a variable, then it displays an error message asking the user to include this classifier in the sequence diagram. Note that the classifier with *q.f* specifies that the object the variable *q* is holding encapsulates this instance of the class *SortAlgorithm*. Such encapsulation specification is only required for the classifiers that do not receive a call. For other classifiers, the encapsulates relation is derived from the activation and focus controls.

- All the names of a classifier should be specified. There is not a strict rule in UML sequence diagrams for specifying the reference variables of a call action.

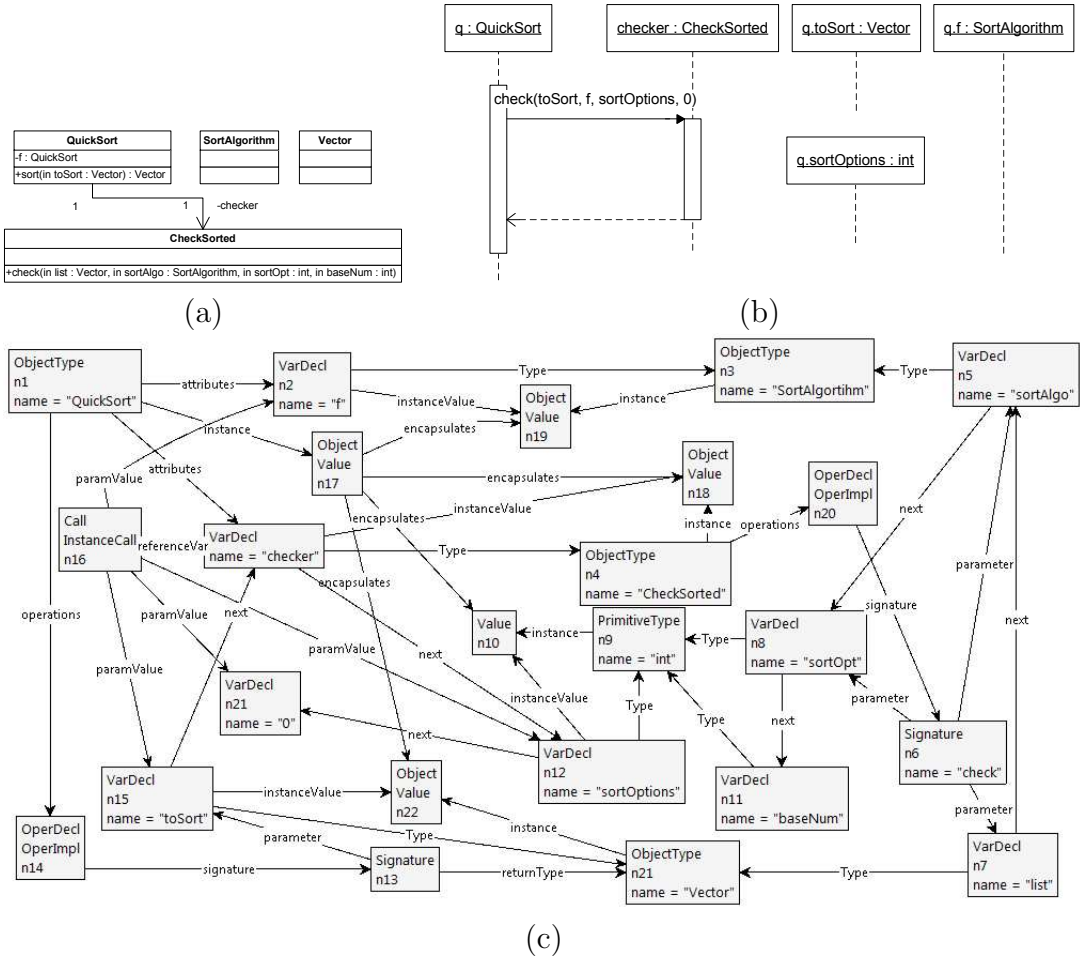


Figure 2.8: a) The class diagram of an example software system. b) a sequence diagram showing the call action from an instance of class *QuickSort* to an instance of class *CheckSorted* that passes a value in the last argument. c) The DCML model of the UML diagrams shown in (a) and (b); the model contains an error because DCML does not support constant values as arguments.

Since DCML supports communication between objects through encapsulation, the reference variables of the call actions play a crucial role during simulation. We overcome this lack of specification by stating that the classifier name is the name of the reference variable (one could be inclined to specify call actions in the form *f.foo()*; however, UML diagram editors do not support this scheme). Following this rule, we programmed the UML-to-DCML converter to use the names of the classifiers to identify the reference variables of the calls. During conversion, the converter tries to find if the name of a classifier is already a declared variable (e.g. it may be an attribute of the class). If a variable cannot be found, then the converter cannot resolve the reference variable and adds a new variable declaration node and a new object node. This behavior causes problems with objects that are passed as arguments. An example case of this is shown in Figure 2.9-(a). Here, an instance of class *QuickSort* passes an instance of the class *SortAlgorithm* through the call action to *checkLess()*. The instance of the class *CheckSorted*, upon receiving the call to it's method *checkLess()*, calls the method *SortAlgorithm.mbar()*. This call is received by the object that was passed in the call action because the call to *mbar()* is received by the same classifier. However, the name of this classifier in both calls is *f*, suggesting that the instance of *CheckSorted* accessed the instance of the class *SortAlgorithm* through a variable named *f*. In reality, the method *checkLess* accesses this instance through its parameter *sortAlgo*. Because a variable name *f* is neither an attribute of class *CheckSorted* nor a parameter of the method *checkLess()*, the converter adds a new variable declaration node as the reference variable of the call action. This behavior can be prevented by specifying all the names other classifiers use to access a classifier as shown in Figure 2.9-(b). In this Figure, the converter would be able to resolve the parameter *sortAlgo* and set it as the reference variable of the call action.

2.2 Execution Semantics and Simulation

The DCM is simulated by automatically triggering the appropriate graph transformation rules that represent the OO-execution semantics of the UML models. We formed a graph production system (a collection of graph transformation rules [77]), consisting of *55* graph transformation rules that model the following OO execution semantics for UML models: method dispatch, parameter passing, returning a value, and object creation. The simulation generates a state-space showing all the executed methods. This section details these 55 graph-transformation rules. However, before going into the details of the models, we first informally describe graph transformation rules and how they are modeled in GROOVE.

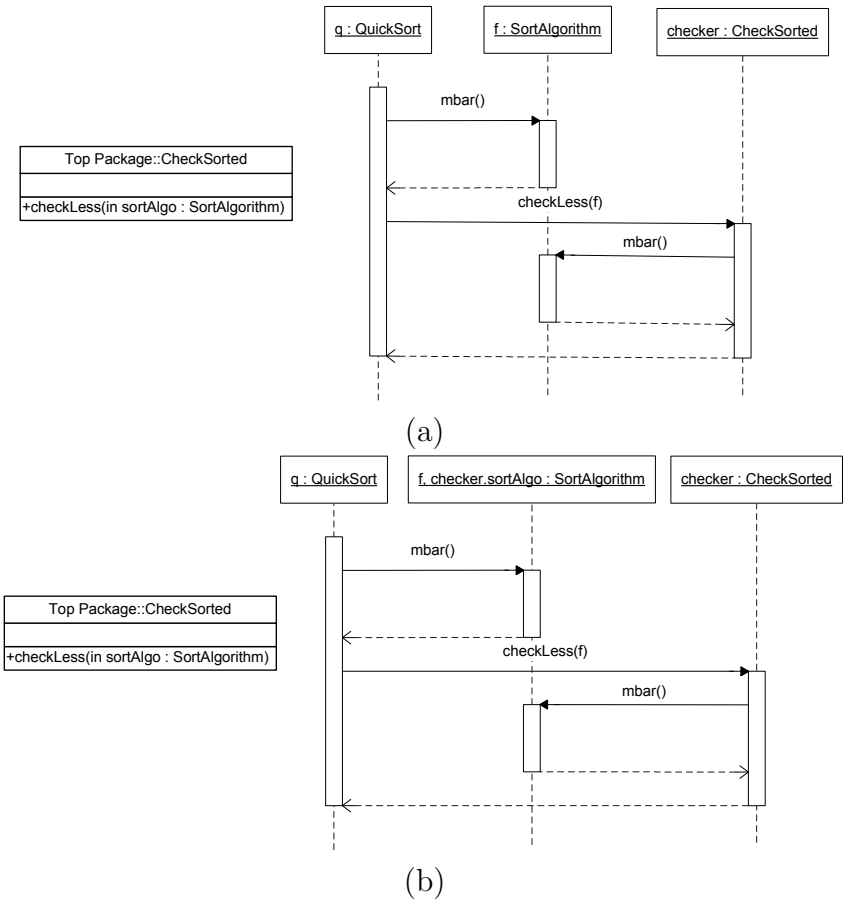


Figure 2.9: a) The class diagram of an example software system. b) a sequence diagram showing the call action from an instance of class *QuickSort* to an instance of class *CheckSorted* that passes a value in the last argument. c) The DCML model of the UML diagrams shown in (a) and (b); the model contains an error because DCML does not support constant values as arguments.

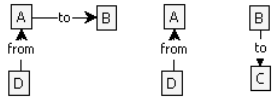
Figure 2.10: An example graph transformation rule that adds the node labeled C

A graph transformation rule has a left-hand side, L, a right-hand side, R and set of negative application conditions N. The rule transforms a source graph G to a target graph H by searching for the occurrence of L in G where N does not occur and, then, replacing L with R to reach H. In order to say L occurs in G all the nodes and edges in L should also be found in G [46]. When L of a transformation rule occurs in G where N does not occur then the transformation rule is said to match; a rule can have multiple matches.

In GROOVE, both left-hand and right-hand side of a graph transformation rule are represented in the same graph. The modifications the rule makes on the host graph are specified using keywords:

- The keyword *new* (or the color green) is used for the edges/nodes that are added. These nodes are not in the left-hand side of the rule but are in the right-hand side of the rule.
- The keyword *del* (or the color blue) is used for the edges/nodes that are deleted. These nodes are in the left-hand side of the transformation rule but are not in the right-hand side.
- The keyword *not* (or the color red) is used for negative application conditions [64]; these edges/nodes should not exist in the part of the host graph where the left-hand side of the transformation rule exists.
- All other edges/nodes are both in the left-hand side and right-hand side of the transformation rule.

An example graph transformation rule is presented in Figure 2.2. For this to rule to match, nodes labeled A and B that are connected to each other with an edge labeled *to* from node A to node B should exist in the host graph. If a node is connected to the node labeled A with an edge labeled *to* then this rule does not match. When applied, this rule removes the edge labeled *to* between nodes A and B and adds a new node labeled C and an edge labeled *to* between nodes B and C. Figure 2.11-(a) depicts a host graph G in which the rule of Figure 2.2 has a match. Figure 2.11-(b) depicts the target graph after this rule has been applied.



(a) (b)

Figure 2.11: (a) An example host graph G. (b) The target graph H, after the transformation rule in Figure 2.2 has been applied.

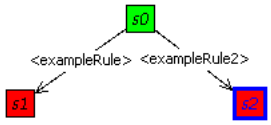


Figure 2.12: An example graph transition system

GROOVE is a state space generator: it takes a graph production system (a set of graph transformation rules) and an initial graph as input. When a graph transformation rule is applied to the graph changes the state of that graph. At a state GROOVE automatically applies all the applicable transformation rules from the graph production system. By applying all the possible transformations at consecutive states, GROOVE generates the state space; each transition is the graph transformation rule applied between the states. We illustrate this by the following example: Assume that the name of the transformation rule presented in Figure 2.2 is *exampleRule* and we have another graph transformation rule named *exampleRule2* that is similar to *exampleRule*; however, this rule adds a node labeled D rather than C. If we select the graph presented in Figure 2.11-(a) as the initial graph I, we have the graph production system $P = \{\{\text{“}exampleRule\text{”}, \text{“}exampleRule2\text{”}\}, I\}$. The GTS of P is presented in Figure 2.2. The initial state is presented with the node labeled $s0$ in this figure and the graph of this state is the graph I. The application of the rule *“exampleRule”*, which shown as an edge labeled with the name of the rule, changes the state from $s0$ to $s1$. The GTS has two branches because both transformation rules can be applied to the initial graph. The states $s1$ and $s2$ are final state of this transition system because neither of the rules (*exampleRule and exampleRule2*) match to the graphs at these states.

As discussed before, GROOVE automatically applies all transformation rules that can be applied at a state from the graph production system. This is called *free-form* application. However, it is possible to give priorities to the rules such that at a state the rules with the highest priority that match to the graph at that state is applied.

In our system, the initial graph is the DCM with a call action ready to be executed, the graph transformation rules model the execution semantics such as calls. Thus, with state space generation, we simulate the execution of the UML models.

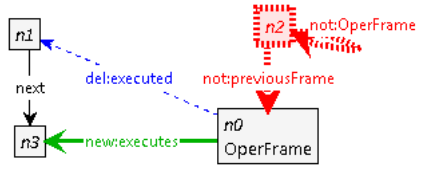


Figure 2.13: The transformation rule modeling the semantics of the program counter.

2.2.1 Program Counter

In DCML, the actions that belong to a method are ordered with edges labeled *next*. The edge labeled *executes* from an operation frame node to an action node shows the action that is being simulated. When the simulation of the action finishes, this edge is replaced (by one of the transformation rules responsible for simulating the action) by an edge labeled *executed*. For example, the simulation of the call action finishes when the called method returns. The transformation rule responsible for deleting the operation frame of the executed method also replaces the edge labeled *executes* by the edge labeled *executed*.

When the current operation frame points to an action node with an edge labeled *executed*, then the transformation rule presented in Figure 2.13 matches. Here, the node $n0$ designates the executing operation frame, the node $n1$ is the executed action and the $n3$ is the action that succeeds the action at node $n1$. The transformation rule moves simulation to the next action by deleting the edge labeled *executed* and adding the edge labeled *executes* to the action that comes after the executed action.

2.2.2 Method Call

A method call requires certain type checks to be enforced at compile-time. UML editors also employ similar checks so that the call is made to a compatible type (for example ArgoUML does not allow one to enter a method that does not exist in the target class of a call action). We assume that these static checks are enforced and the call action is valid. Because DCML only supports communication between objects through encapsulation, the execution semantics of method calls are very similar to run-time evaluation of method invocations of Java (See section 15.12.4 at [9]). Method invocation consists of finding the latest implementation of the method in the inheritance hierarchy and passing the arguments that are executed in the following manner: 1) calculating the type of the object the reference variable is holding; that is, the reference type of the call 2) starting from the reference type traversing the inheritance hierarchy upwards until an object-type that declares the

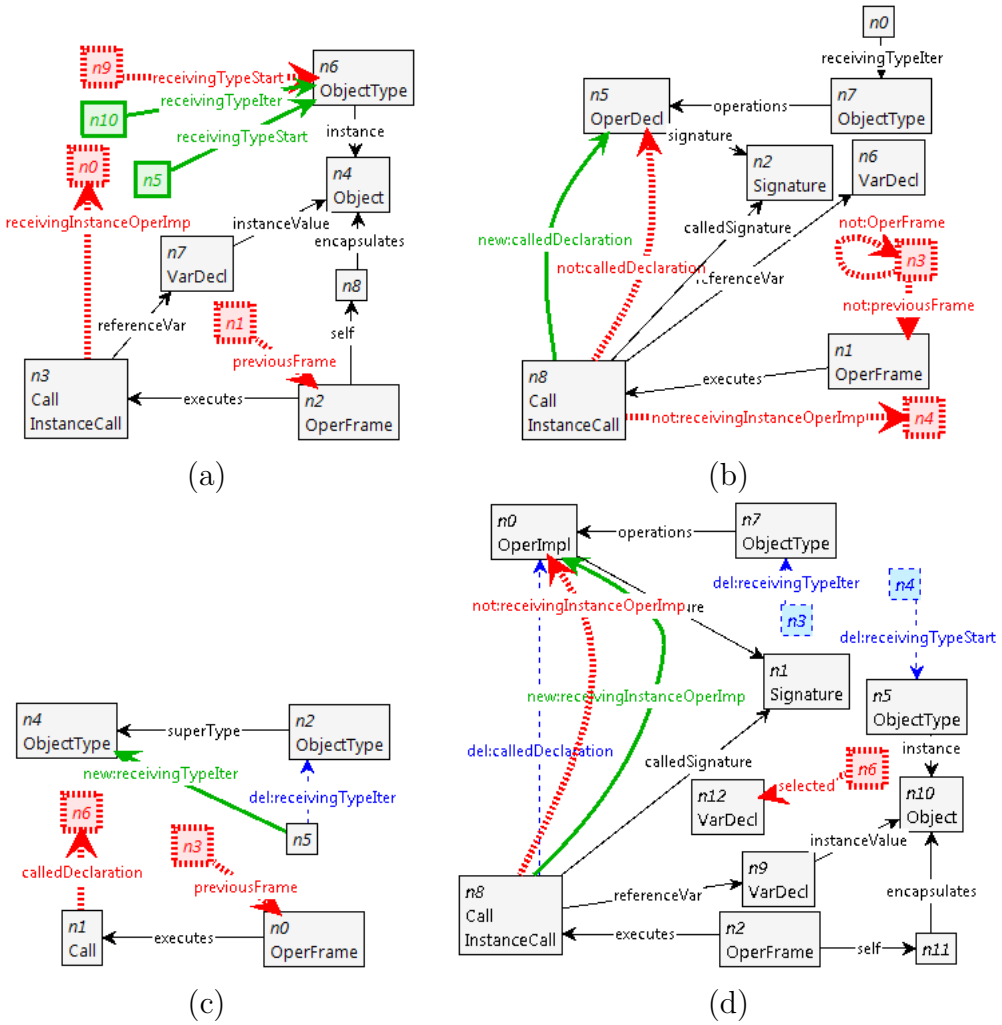


Figure 2.14: Graph transformation rules for finding the newest implementation of the called method: (a) calculates the target reference type and marks it (b) finds the latest declaration of the method (c) moves the mark up one level in the inheritance hierarchy, (d) checks whether the latest declaration implements the method.

method is found 3) passing the arguments 4) checking that the latest declaration implements the method. From these, the semantics of parameter passing is described in Section 2.2.7. This section describes the 4 transformation rules that are used for finding the latest implementation of the method:

1. The rule in Figure 2.14-(a) is used for finding the reference type of the call. In sequence diagrams, the reference variables of the call actions are only variable declarations. So, it is sufficient to find the object-type whose instance this reference variable is holding to identify the reference type (i.e. there is no need to execute nested calls). In this figure, the reference variable is node $n7$ (i.e. the variable declaration node that is connected to the call node with an edge labeled *referenceVar*) and the object it is holding is node $n0$. For this rule to match, the reference variable's value at the current operation frame should be an object and this object should be connected to an object-type node with an edge labeled *instance*. If, for example, the reference variable does not hold an object, then the call cannot continue. This is equivalent to a *null pointer exception*. The transformation rule adds two nodes and edges. From these, the edge labeled *receivingTypeStart* marks the object-type from which the traversal in the inheritance hierarchy starts. The edge labeled *receivingTypeIter* marks the object-type that is traversed. Since the reference type of the call is the type the traversal starts from and since it is the first type to be traversed, these edges are connected to the object-type node that is the reference type of the call.

2. The rule in Figure 2.14-(b) marks the latest declaration of the method. If the traversed object-type contains an operation declaration node that has the same signature as the called signature then this operation declaration is the latest declaration of the method. In the depicted transformation rule, the traversed object-type node is node $n7$ and the called signature is node $n2$. The rule matches when the traversed type has an operation declaration node ($n5$) that is connected to the same signature node as the called signature. The rule marks the declaration by adding an edge labeled *calledDeclaration* between the call node ($n8$) and the operation declaration node.

3. If the object-type traversed (i.e. the edge labeled *receivingTypeIter* pointing to) does not have the method declaration then its super-type should be traversed. The transformation rule in Figure 2.14-(c) deletes the edge labeled *receivingTypeIter* and adds another edge with the same label pointing to the super-type of the traversed object-type. This rule has a lower priority then the rule presented in the previous step; these two rules do not match at the same time. In this way, if the traversed object-type has the method declaration

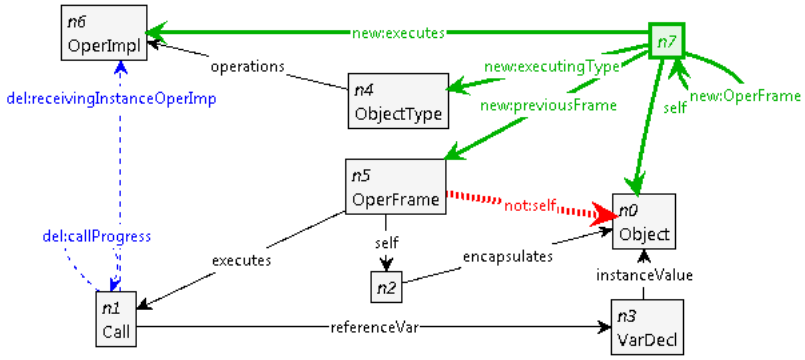


Figure 2.15: Graph transformation rule that dispatches the method after the object-type that implements the method is discovered by the rules presented in Figure 2.14.

then the traversal stops because the rule in the previous step deletes the edge labeled *receivingTypeIter* which is required for this rule to match.

4. After finding the method declaration and preparing the arguments, the method can be dispatched. However, before dispatching, we must be sure that the method is implemented. The transformation rule in Figure 2.14-(d) matches when the method declaration node marked in step 2 is also an method implementation node (i.e. that is also labeled *OperImpl*). When this rule matches, it marks the method implementation to be ready for dispatch by adding the edge labeled *receivingInstanceOperImpl*. The previous rule could also be modeled so that the traversal would search for the operation implementation. However, we made this a separate transformation rule because at run-time parameter passing is done after the operation is located and before the method is dispatched.

After the object-type that implements the method is discovered, the method can be dispatched. The graph transformation rule presented in Figure 2.15 dispatches the method. Here, the dispatching is done by creating a new operation frame node ($n7$), that is connected to the dispatched method (*OperImpl* node) with an edge labeled executes. The *self* of the new frame is the object to which the call is made; thus, the rule adds the edge labeled *self* between the newly added frame ($n7$) and the object the reference variable holds ($n0$). The executing type of the new frame is the object-type that implements the method ($n4$). The frame where the call is initiated from is connected to the new frame with an edge labeled *previousFrame*. With this edge, the frame that will be returned when the execution of the called method finishes is marked.

2.2.3 This-calls

A *This-call* is special type of call that does not have a reference variable and the executing object is always the same as the calling object. A this-call is dispatched in the following steps: 1) if the current executing object-type (i.e. the object-type node that implements the method currently executing) implements the called method prepare it for the method for dispatch. 2) Otherwise, iterate through the inheritance hierarchy until the object-type that implements the method is found. These steps are realized by 4 transformation rules presented in Figure 2.16. Here, the transformation rule presented in Figure 2.16-(a) marks the object-type whose instance is executing (i.e. it is the *self* of the operator frame). Here, the node $n6$ is the operation frame that is currently executing. The traversal of the inheritance hierarchy should start from the the current executing object-type; this object-type implements the method that the call is made from. Following the edge labeled *executingType* from the operation frame node, the executing type can be found. In the figure, it is the node $n7$. When applied, the rule adds the edges labeled *receivingTypeIter* and *receivingTypeStart* pointing towards this node.

If the currently traversed object-type (i.e. the object-type node the edge labeled *receivingTypeIter* points to) has an operation declaration node that has the signature the call specifies, the rule in Figure 2.16-(b) matches. Similar to the rule presented in Figure 2.14-(b), it marks the object-type by adding an edge labeled *calledDeclaration*. If the currently object-type does not have an operation implementation that has the signature the call specifies, then the rule in Figure 2.16-(c) matches. This rule moves the traversal one level up in the inheritance hierarchy. It is important to note here that the rule presented in Figure 2.16-(b) has a higher priority than the rule presented in Figure 2.16-(c). The edge labeled *calledDeclaration* is a negative application condition for the rule presented in Figure 2.16-(c). Because the rule in figure 2.16-(b) adds this edge, the rule in Figure 2.16-(c) matches to the host graph until (b) matches. In this way, the iteration through the inheritance hierarchy is achieved.

When the object-type implementing the method is found, the rule presented in Figure 2.16-(d) matches, which simply marks the operation for dispatch. The actual dispatch is done by the graph transformation rule presented in Figure 2.17. The main difference between the rule presented in this figure and the dispatch rule used for method calls (Figure 2.15) is that this rule adds an operation frame node that has the same self as the operation frame node where the call is made. The added operation frame is the node $n2$ and its self is the node $n4$. This object node is also the self of the operation frame (node $n0$) where the this-call is made.

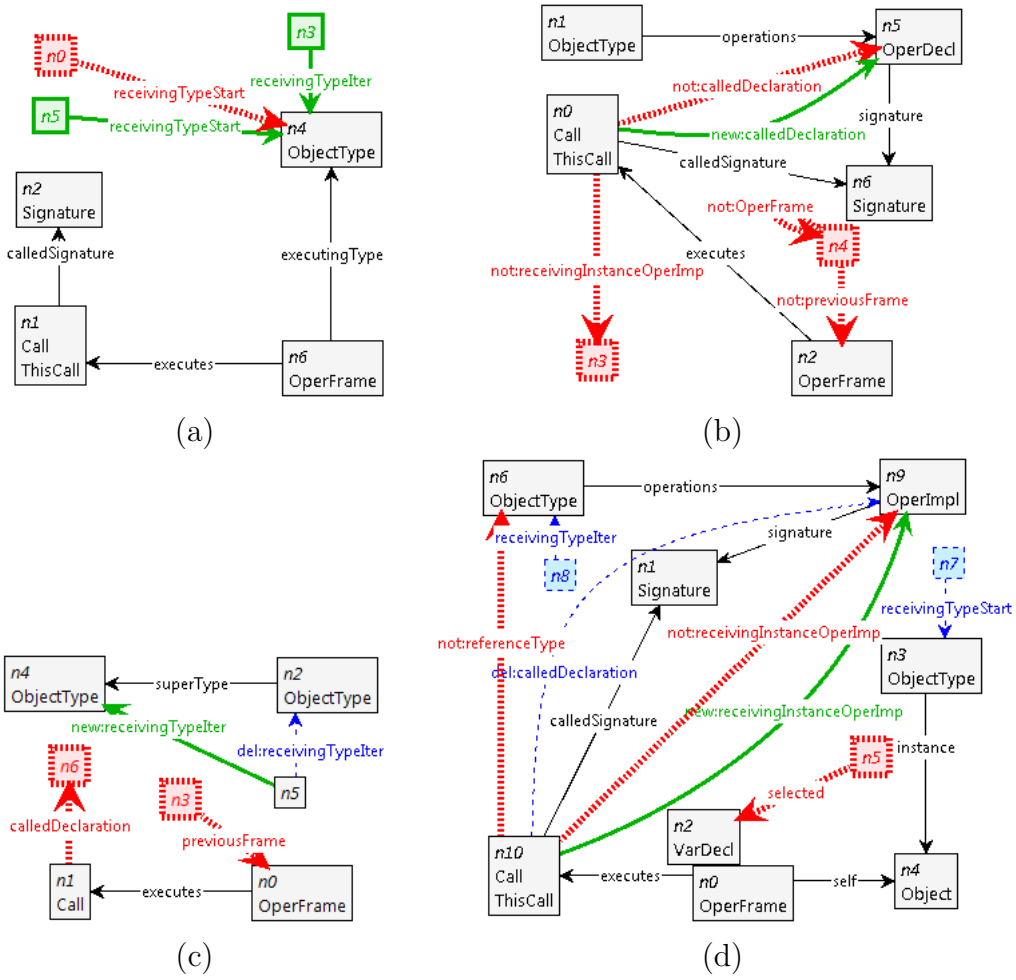


Figure 2.16: Graph transformation rules modeling the execution semantics of a this-call: (a) finds the object-type that declares and implements the method, (b) iterates through the inheritance hierarchy (c) marks the object-type that is going to receive the call, (d) prepares the method for dispatch.

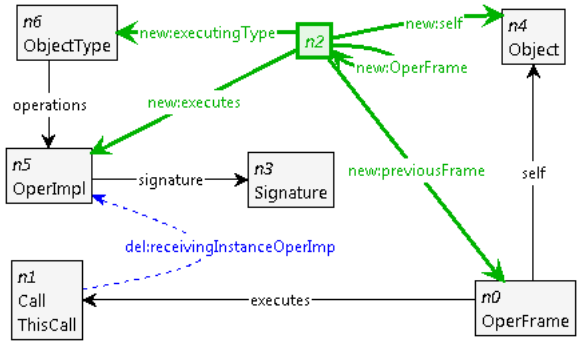


Figure 2.17: Graph transformation rule that dispatches the method for a this-call.

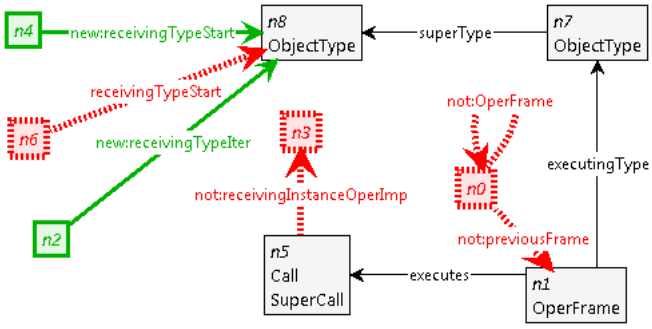
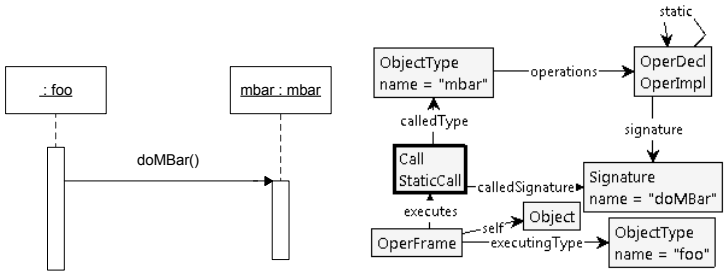


Figure 2.18: Graph transformation rule for marking the super-type of the current executing object's object-type. This rule is used during a super-call.

2.2.4 Super-Calls

Super-calls are similar to this-calls; they are both calls without a reference variable. The main difference between these calls is that in super-method calls the iteration in the inheritance hierarchy starts from the super-type of the current executing-type, the object-type node connected to the operation frame with an edge labeled *executing-type*. This ensures that even though the currently executing object's object-type overrides the called method, the overridden implementation of the method is called.

When execution reaches a super-method call, i.e. a node labeled *SuperCall*, the transformation rule presented in Figure 2.18 matches. Here, the executing-type is the node $n7$. Note that the edges labeled *receivingTypeIter* and *receivingTypeStart* are added pointing to the super-type of the executing type (node $n8$). In this way, the traversal starts from the super-type of the executing object-type. The rest of the super-method call is executed similarly to the this-call: 1) if the currently marked type implements the method, the method is marked. 2) if the currently marked



(a) (b)

Figure 2.19: a) a sequence diagram with a static call action, the method *mbar.doMBar* is a static method, and b) its equivalent DCM.

object-type does not implement the method, traverse the inheritance hierarchy until the method implementation is found. 3) The method is dispatched by adding a new operation frame that has the same *self* as its previous frame and the executing type is always an object-type that is at a higher level in the inheritance hierarchy than the executing type of the previous frame.

2.2.5 Static-Method Calls

In UML sequence diagrams, static calls are represented by a call action to a classifier whose name and class name are the same; that is, rather than referring to an object the call action refers to a class. The static calls are modeled in DCML with call action nodes that are also labeled *StaticCall*. The difference between a method call and a static call is that a static calls refers to an object-type rather than to a variable. Figure 2.19-(a) presents a sequence diagram with a static call that refers to the class *mbar* and calls the signature *doMbar*. This call in DCML is presented in Figure 2.19-(b). In this figure, the call action node is the emphasized node. This node is connected to the object-type node *mbar* by an edge labeled *referenceType* to show that the static call refers to the object-type *mbar*. The object-type *mbar* has an operation implementation that is also labeled static; in DCML static operations are represented by operation implementation nodes that are also labeled static. This operation's signature is *doMbar()* because the operation implementation node is connected to a signature node with the same name and does not have any parameters or return values. The edge labeled *calledSignature* from the static call node is connected to this signature node to show that the signature *doMBar()* is called.

The execution semantics of a static-method call is captured in the 3 transformations depicted in Figure 2.20. The first transformation rule, Figure 2.20-(a), matches when

an execution reaches an action labeled *StaticCall*. In this rule, the static-call is the node $n4$ and the object-type that is referred by this call is the node $n2$. The referred object-type should declare a static method that has the same signature as the called method. Thus, this rule matches when the referred object-type (node $n2$) has a static operation declaration that is connected to the signature node to which the call node is connected. In the rule, this operation declaration node is the node *n0*. In DCML, static methods are represented by operation implementation nodes that are also labeled as *static*. If the referred type has a static operation declaration that has the same signature as the called signature, then this rule adds an edge labeled *calledDeclaration* to mark the static operation declaration. Otherwise, the static method does not exists in the object-type referred to by the call and the call is not permitted to go on (i.e. the rule in Figure 2.20-(a) does not match and the simulation stops at the call node).

After passing the arguments to the static operation, the rule in Figure 2.20-(b) matches. This rule checks if the static operation declaration is also implemented. If so, this rule adds an edge labelled *receivingInstanceOperImp* to mark the operation implementation node. The actual dispatch of the method is done by the rule presented in Figure 2.20-(c). After the operation implementation node is marked, this rule matches and adds the operation frame for the static method.

Here, the node $n4$ represents the operation frame node added by the rule. The self and executing type edges are connected to the referred object-type. The operation frame nodes for static methods have an object-type node as it's self. The DCML meta-model shows that an object-type node can be connected to value nodes by edges labeled *encapsulates*. This shows the instance/values the static methods of the type can refer through static attributes. The resolution of these static values is done by following the *self* edge of the operation frame.

2.2.6 Object Creation

In UML sequence diagrams, object creations are shown by create actions. The DCML equivalents of create actions are call nodes labeled as *CreateOper*. Certain UML sequence diagram editors allow to specify the constructor called by the create action. Constructors in DCML are represented by operation implementation nodes that are also labeled as *Constructor*. The constructor that is called by the create action is presented by an edge connecting the node labeled *CreateOper* to a signature node labeled *calledSignature*. If a constructor is defined for the create action, the UML to DCML converter adds this edge to the constructor. If, on the other hand, a constructor is not defined, then the converter adds a default constructor (i.e. it takes no parameters). The call actions that come after the create actions are in the focus

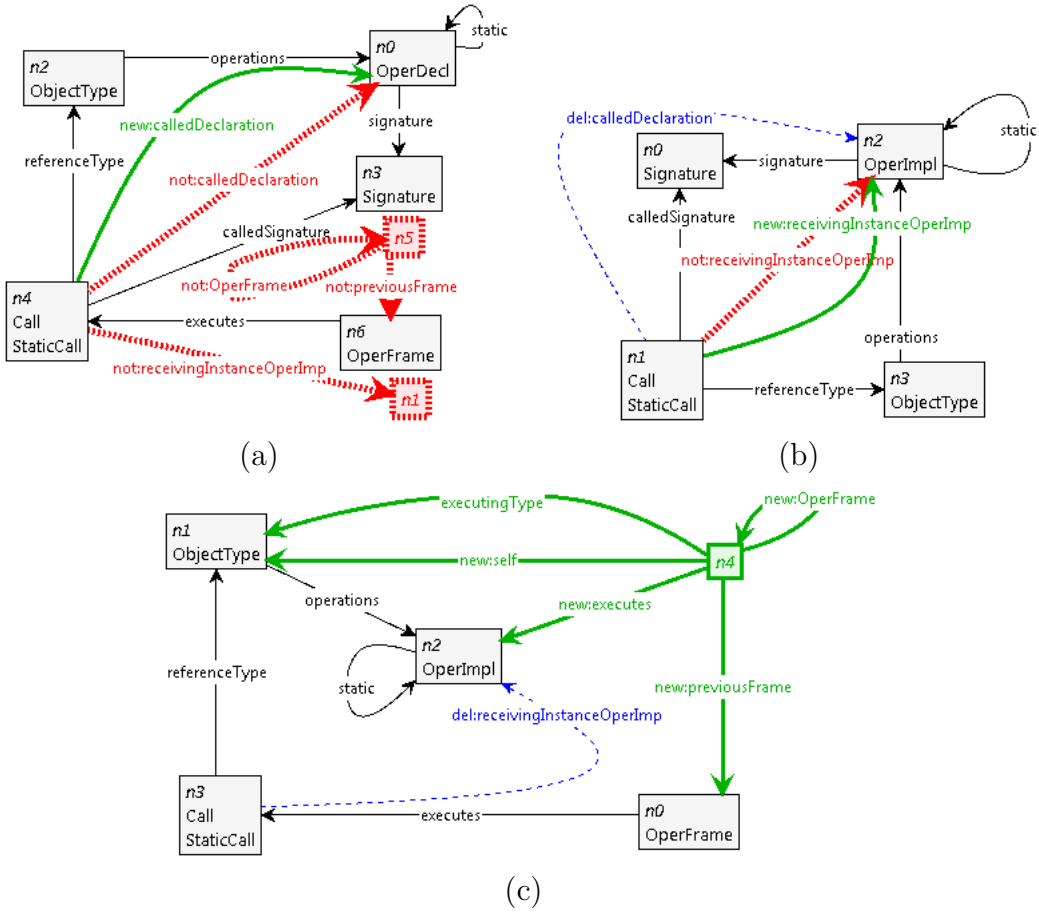
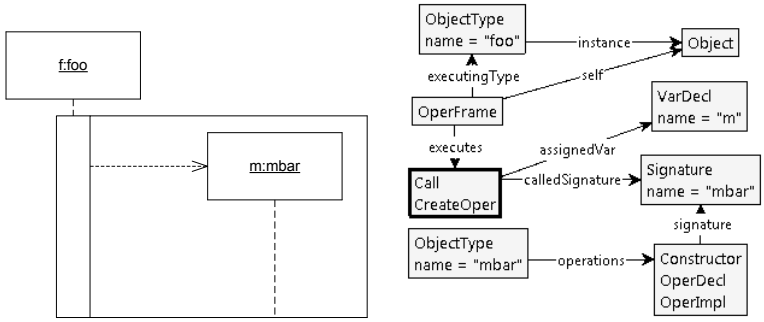


Figure 2.20: Graph transformation rules modeling the execution semantics of a static method call: (a) checks if the referred object-type declares the static method, (b) prepares the static method for dispatch (c) dispatches the static method.



(a) (b)

Figure 2.21: a) a sequence diagram with create action b) its equivalent DCM

control of the constructor; so, the converter adds these actions as the statements of the constructor in DCML.

In Figure 2.21-(a) a sequence diagram with two objects is presented where the object labeled *f* creates an instance of the class *mbar*. Figure 2.21-(b) presents the DCML equivalent of this sequence diagram. Here, the create action is the emphasized node. The created instance of the class *mbar* is labeled *m* in the sequence diagram. Because DCML supports communication between object through encapsulation, *m* is represented as a variable (i.e. variable declaration node). The create action node (the emphasized node) is connected to this variable declaration node by an edge labeled *assignedVar* to show that the variable *m* will hold the created instance of class *mbar*. The create action node is also connected to the signature node labeled *mbar* by an edge labeled *calledSignature*. This signature is the signature of the constructor; that is, a constructor of the object-type *mbar* is connected to this signature node. Since the sequence diagram does not specify the constructor called by the create action, a default constructor is created by the UML to DCML converter.

The semantics of object creation is captured with 3 graph transformation rules: one is used for creating the object and the others are used for dispatching the constructor. The transformation rules used for dispatching the constructor are very similar to the transformation rules presented in Figure 2.14-(d) and Figure 2.15. The transformation rule that creates the object is presented in Figure 2.22. In this figure, the node $n0$ is the created object. This object is connected to the *self* of the operation frame by an edge labeled *encapsulates*. The *self* of the operation frame, node $n7$, is an unnamed node because the self can be an object-type (if the constructor is called from a static method) or an object. The new object is connected to the create action by an edge labeled *passedReturnValue*. After the constructor call finishes, this object value should be assigned to the variable declaration node

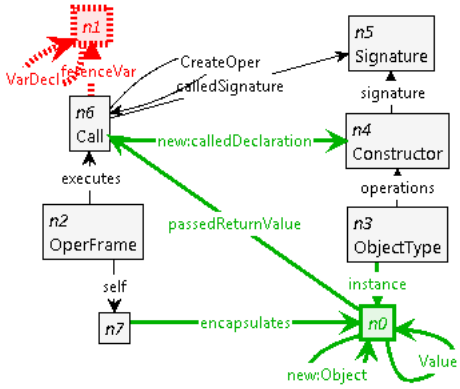


Figure 2.22: Graph transformation rule that creates an object and prepares the execution of the call to the constructor.

that is connected to the create action node by an edge labeled *assignedVar*.

The transformation rule also checks if the create operation calls a constructor; for the rule to match the called signature should be a signature of a constructor (node $n4$). The edge labeled *calledDeclaration* is added by the rule to mark the constructor declaration. Similar to method calls, after this rule the arguments are passed, then it checks whether the constructor is implemented. Finally, the constructor is dispatched.

2.2.7 Parameter Passing

In DCML, parameters of a method are represented by variable declaration nodes that are connected to the signature nodes by edges labeled *parameter*. The edge labeled *next* is used for ordering the parameters of a signature. The arguments a call action passes to a method are also represented by variable declaration nodes. However, these variable declaration nodes are connected to the call action node by edges labeled *ParamValue*. The edges labeled *next* are used for ordering the arguments as well.

The passing of arguments is done at OO run-time when the method declaration is discovered. For each parameter the method takes, the value of the argument at the current frame is computed and, then, this value is passed. The semantics of parameter passing is modeled with 4 graph transformation rules presented in Figure 2.23. Note that these rules do not cover the static-type checking done at compile time. The first transformation rule selects the first parameter of the method and the first argument the call passes. The selection is done adding the node $n2$

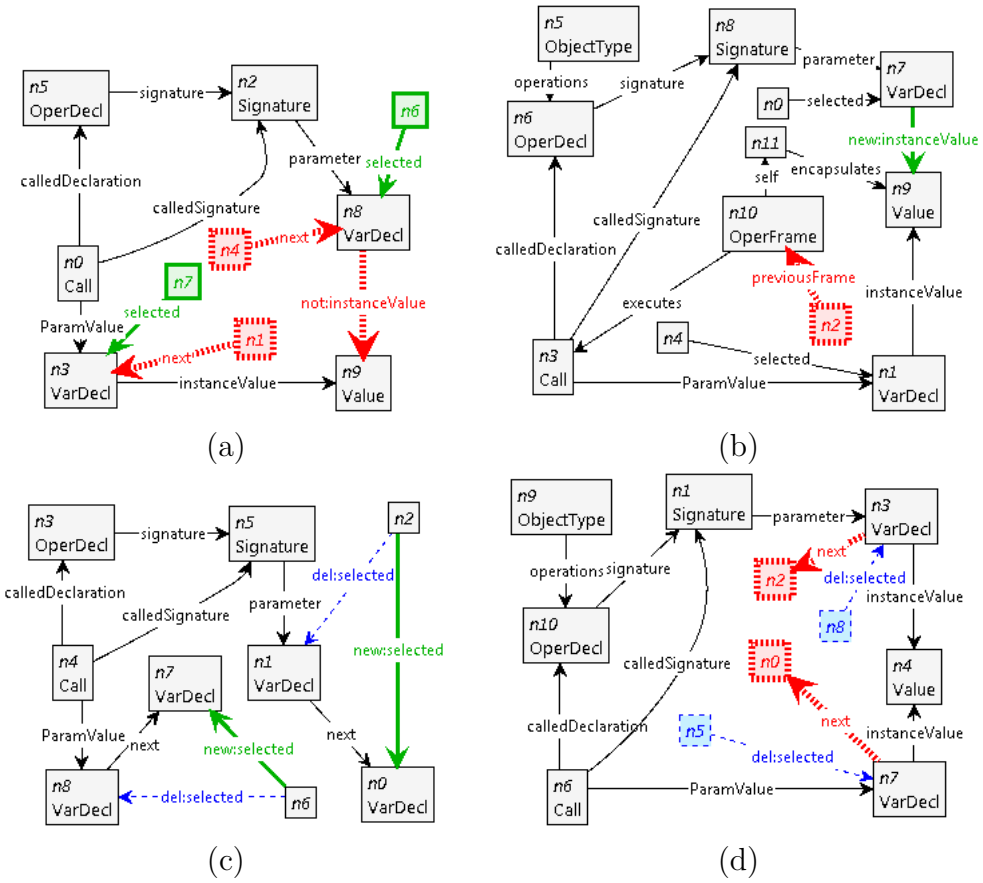


Figure 2.23: The graph transformation rule modeling the semantics of parameter passing: a) Selects the first parameter and argument b) passes the value of the selected argument to the method declaration c) moves to the next parameter d) completes the parameter pass operation.

and an edge labeled *selected* from this node to the first argument of the call. The node $n1$ that is connected to an argument of the call (node $n0$) by an edge labeled *next* is a negative application condition. For the rule to match, there should be at least one argument before which there is not an argument. Since this argument is the first argument, this rule always selects the first argument. Using a similar negative application condition (node $n4$), the rule also selects the first parameter of the method signature (node $n2$).

The second rule (Figure 2.23-(b)) is used for finding the value of the selected argument in the current frame. The node $n10$ represents the current operation frame and the variable declaration node $n1$ is the selected argument. Following the edge labeled *self* from the current frame, the type or the object that is executing can be identified; this is the node $n11$. The value of the argument in the current frame is identified by following the edge labeled *encapsulates* and the edge labeled *instanceValue*, which is the node $n9$. Thus, for this rule to match, a value for the argument should exist in the current frame. The rule also passes the value of the argument to the parameter (node $n7$) by adding the edge *instanceValue*.

The third rule (Figure 2.23-(c)) iterates to the next argument and parameter. This rule deletes the edge labeled *selected*. With the edges labeled *next* the next argument and parameter is identified (nodes $n7$ and $n1$). So the rule adds an edge labeled *select* pointing towards these. When these are no more arguments and parameters to iterate, the rule presented in Figure 2.23-(d) matches. The negative application conditions, nodes $n2$ and $n0$ with incoming the edges labeled *next*, force the rule to match to arguments and parameters after which there are not any arguments and parameters. This rule simply deletes nodes and edges used for iterating over the arguments and parameters. Note that the rules presented in Figure 2.14-(d) and Figure 2.16-(d) only match when these nodes and edges are deleted. If, for example, there is a problem with the parameter pass then the edge labeled *selected* does not get deleted, then the method dispatch cannot continue.

The rule presented in Figure 2.23-(b) sets the values of the parameters by adding the edge labeled *instanceValue*. However, before the method starts executing these values should be encapsulated by the executing object-type or object. The value of a variable in the executing frame is resolved with the edges labeled *encapsulates* and *instanceValue*. Because the passing of the values to the parameters is done before creating the operation frame of the method to be executed, none of the transformation rules described above add the edge labeled *encapsulates*. Thus, after the operation frame is created, the edge labeled *encapsulates* should be added from the *self* of the new frame (i.e. the executing object-type or object) to the value of each parameter. The transformation rule presented in Figure 2.24 is used for this purpose and it is applied right before the first action of the executing method.

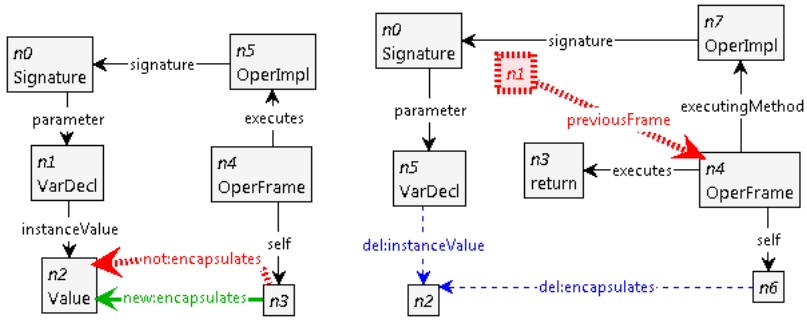


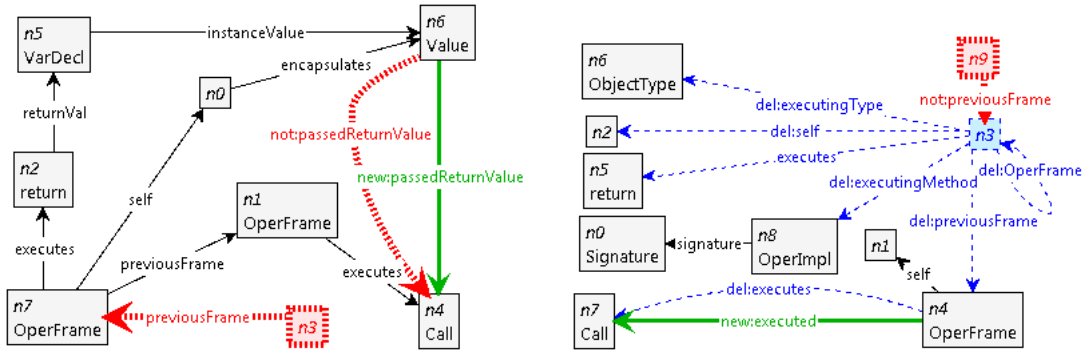
Figure 2.24: a) With this transformation rule, the values of the parameters are encapsulated by the executing type or the object. b) this rule deletes the encapsulated parameter values, so the object or object-type cannot access them after the method returns.

The values of the parameters should not be accessible to the object or the type after its method finishes execution. This is achieved by the transformation rule presented in Figure 2.24-(b). This rule matches when the current frame (node $n4$) reaches the return action (node $n3$) and when the executing method has parameters. The rule deletes the edges labeled *instanceValue* and *encapsulates*. In this way, the object or the object-type whose method finished executing cannot access the values passed to the method. Note that these values remain accessible by the frame to which the method will return.

2.2.8 Return Action

The return actions are represented in DCML by nodes labeled *return*. If the method returns a value, then the return action in the sequence diagram has an argument. This argument's value is the value returned by the method. In DCML, the return argument is represented by a variable declaration node that is connected to the node representing the return action by an edge labeled *returnVal*.

The semantics of return actions are captured with the two transformation rules presented in Figure 2.25. The transformation rule in Figure 2.25-(a) matches when the execution reaches a return action node that has a return value. This rule is used for passing the return value of the method to the previous frame (i.e. the frame from which the call to the returning method is made). The node $n2$ is the return action node and the variable declaration node $n5$ is the variable that stores the return value. The current frame is the node $n7$ and the frame that will be returned to is the node $n1$. The frame that will be returned to is connected to a call action node, node $n4$,



(a) (b)

Figure 2.25: a) The transformation rule that passes a reference to the return value b) The transformation rule that deletes the executing frame and returns the previous frame.

by an edge labeled *executes*; this shows the call action that called the method that is returning. The *self* of the current frame is designated with node $n0$. Note that the rule does not specify a label for node $n4$, because the *self* can be an object-type or an object. Following the edges labeled *encapsulates* and *instanceValue*, the value that is returned is resolved. The return value is passed by adding an edge labeled *passedReturnValue* from the value node to the call action node.

The transformation rule depicted in Figure 2.25-(b) is used for deleting the frame that finished execution. Here, the frame that finished its execution is the node $n3$. The rule deletes this node and, thus, all the edges from this node are also deleted. Following the edge labeled *previousFrame* from the node $n3$ (the returning frame), the frame that will continue its execution can be identified; this frame is the node $n4$. The call action that made the call to the returning method is the node $n7$. Note that the rule also deletes the edge labeled *executes* between the frame node $n4$ and the call node $n7$ and adds an edge labeled *executed* between these nodes. This states that the call action has finished executing and the program counter can move to the next action. This transformation rule has a lower priority then the rule presented in Figure 2.25-(a). So, if the method has a return value then first the return value is passed to the previous frame and then the frame is deleted.

The transformation rule used for returning the return value of a method to the frame where the method is called (Figure 2.26), neither assigns the value to a variable nor adds the *encapsulates* edge. The assignment is executed after the method returns and the return value only gets assigned to a variable when the call action specifies a variable to assign the return value to. In sequence diagrams, the variable that gets the return value either can be shown in return action or in the call action. Our UML

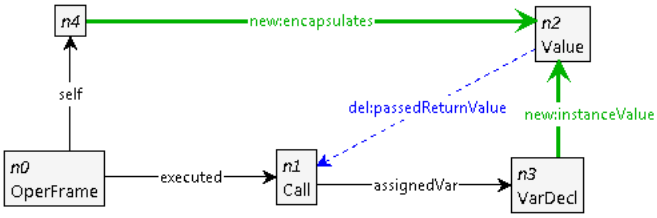


Figure 2.26: The transformation rule used for assigning the return value

to DCML converter accepts both ways of specifying the return value. In DCML, the variable that gets the return value is modeled by connecting the call action node to the variable declaration by an edge labeled *assignedVar*.

Figure 2.26 depicts the transformation rule that is used for assigning the return value to the variable specified by the call action. Here, node $n2$ is the return value, the variable that gets this value is the node $n3$ and the *self* of the operation is the with node $n4$. The assignment is done by adding the edges labeled *instanceValue* and *encapsulates* respectively from the variable declaration node that gets the value and from the *self* of the executing frame. This transformation rule has a higher priority then the program counter transformation rule; thus, if the call specifies a variable to assign the return value, first the value is assigned and then the execution moves to the next action.

2.3 Evaluation of the Execution Semantics

Object-oriented design patterns [59] heavily rely on polymorphism for decoupling the object that receives the call from the object that makes the call. Most design patterns use the execution semantics described above, they can be easily modeled in UML and the runtime behavior of the design patterns is documented (or well-known). As a result, design patterns are good test cases for the execution semantics described in this chapter. Table 2.1 lists the design patterns used for testing the execution semantics with the semantics tested for each design pattern. For these patterns, we simulated the execution of the sequence diagrams and compared this execution with the documented behavior. The simulation and the documented behavior agreed for all design patterns.

Table 2.1: The design patterns used for evaluating the modeled execution semantics

Design Pattern	Remarks about design	Covered Execution Semantics
Strategy	2 strategies are used	Method call, parameter and return value passing
Decorator	5 decorators are used	Method call, super-call, parameter and return value passing
Abstract Factory	2 factory classes are used	Method call, object creation, return value passing
Template Method	2 template classes	This-call, Method call, parameter passing
Proxy Pattern	1 Proxy class, 2 implementors	Method call, parameter passing
Singleton		Static-call, method creation, return value passing

2.4 Related Work

UML models provide a high-level overview of the different aspects of the software. Although the meta-model of UML is documented and the modeling language is widely known, the lack of formal semantics makes it hard to reason about the models. In the literature, formal semantics for different types of UML diagrams are proposed. Use cases capture the software system's behavior from the stakeholders' view; they are mainly modeled using UML use case diagrams and explained in text. This makes it hard to validate/simulate the use case models. Whittle [127] provides formal semantics to use case charts so that use case scenarios can be specified. A use case chart is a three-level diagram: the first level is an activity diagram where the nodes are the use cases, the second level is also an activity diagram where the nodes are scenarios of a use case in the previous level and the third level is the interaction diagrams of the scenarios of the second level. Because the use case charts are formally specify the scenarios, the charts can be simulated. For this, an hierarchical state machine synthesis algorithm is proposed [128] and the tool UCSIM that executes this algorithm and simulates the generated state machines has been developed [72].

Graph transformations have been used to specify formal execution semantics to UML state-charts [85] [86]. For example, Kung et al. [85] generate the graph grammar that models the execution semantics for a given state-chart. These semantics; however, work only on providing verification/visulization for single state-chart.

Dynamic meta modeling is also proposed as a way to add operational semantics to

the UML diagrams [48]. In this approach, the meta model of the UML class diagram is extended with a dynamic meta model that uses the collaboration diagram notations. The state-chart diagrams specify the behavior of the system; for example, in order to trigger a transition a method has to be called which in turn can trigger another event in the state chart of the called method. Using graph transformations the operational semantics such as state transition, method call trigger are modeled. Using a state space generator such as GROOVE and these graph transformations [49], it is possible to simulate the behavior of the software system and generate the state space of the system. Then, the requirements of the system can be verified in the generated state space.

Kleppe et al. [112] provide formal semantics with typed graphs and graph constraints for UML class and diagrams. The main focus of these semantics is increasing/verfiying the quality of UML models. They are focused on correctly creating object diagrams that conforms with class diagrams and UML standards. For example, with these semantics it is possible to find wrong instantiations of the association relations between classes. Compared to our execution semantics, these semantics are *static* and do not model execution of object-oriented software systems. Whereas our graph transformations model the execution of object-oriented software systems such as method dispatches and parameter passing.

The main difference between the approaches presented in this section and our semantics is that we provide semantics that are close to actual object-oriented software execution. Thus, we can simulate and reason about polymorphism. Moreover, the semantics we provide are generic can be applied to any sequence diagram.

Programming languages have well-defined syntax but their execution semantics are informally specified. To formalized the execution semantics of OO programs Kastenberg et al. [76] model execution semantics of the TAAL language (a simplified version of Java) as graph transformations. Here, the idea is that a program in the TAAL language can be compiled into a graph model and simulated using the graph transformation modeling the execution semantics. By using graph-based model checking, the properties of the execution verified. On contrary to the execution semantics we defined for UML models, the execution semantics provided this by are a full-semantic representation and they require a full program to work. Due to differences in the abstraction level the abstraction levels these semantics are not suitable for simulating UML models; mostly, due to the fact that a full representation of the program (and all the input values) may not be available in the OO-design phase. Besides these, a full-semantic representation simulates all the statements of the program which in turn generates large state-space. Large state-spaces can be a problem when the focus is only on the interactions of the objects; a state-space including only the interaction between objects can provide better guidelines on errors

related to the design. The UML models are widely accepted means for modeling the interactions between object and, thus, we choose to define execution semantics only for the actions that can be modeled with this modeling language.

Chapter 3

Verifying Runtime Reconfiguration Requirements on UML models

Runtime reconfiguration allows the decomposition of the software system to be adapted to the environment [107]. With the increased use of software technology in embedded systems and due to a large diversity of client's needs, many devices today are designed to be runtime reconfigurable. For example, many medical devices such as MRI (Magnetic Resonance Imaging) systems are configured at the client site during the installation phase. Additionally, such devices are reconfigured at runtime to optimize these machines with respect to the client's needs.

Runtime reconfiguration is achieved by modifying the interactions between software entities. A runtime reconfigurable software system, usually, consists of two parts: the *application* and *the configuration system*. The application includes all the software entities related to the functionality of the software system. The configuration system, on the other hand, is responsible for detecting the changes in the environment. The configuration system adapts the application through runtime reconfiguration mechanisms. These mechanisms are programming techniques that allow modifications to the interactions between software entities and are, usually, specified in the application. Here, the application requires special attention because one needs to verify that the modification of the interactions due to reconfiguration does not violate the invariants of the application and that the application supports all the desired modified interactions.

Such verifications are very hard to conduct on the implementation of the software because one needs to abstract from the implementation to focus only on the interac-

tions between software entities. This is especially hard for object-oriented software systems (OO) due to complex class hierarchies. Correcting design errors at the implementation phase can also be too costly in case the verification fails and, thus, verification of these requirements on the models of the software before the implementation is more beneficial. UML is a standardized and widely accepted modeling language for detailed OO designs. From these models, the class and sequence diagrams are at a sufficient abstraction level to capture the effects of the reconfiguration on the interactions and because they are at a higher level of abstraction than the implementation, changing these models is not as costly as changing the implementation. These two characteristics make these diagrams good candidates for pre-implementation evaluation of reconfiguration requirements. However, the verification of the reconfiguration requirement is still hard with these diagrams because the designers still have to manually trace through complex class hierarchies, many conditional blocks and, possibly, many sequence diagrams.

In the literature, much attention is given to modeling/specifying how the configuration system reconfigures the application on software component models [107, 88], as wrong reconfiguration specifications may introduce errors in the application. To ensure that the configuration system reconfigures the components at the correct states, approaches for formal specification of the reconfiguration in software architecture models [125, 13, 131] are proposed. Product line variability models for specifying reconfigurable components [130, 66] are also explored. Recently, approaches that include application invariants in modeling the configuration system are proposed [103]. However, these approaches do not provide verification of the specified reconfiguration on the application. Even though the reconfiguration specification is correct, the application may still fail due to wrong realization of the interactions between application components.

Verification of the reconfiguration requirements on the models of the application requires one to reason about how the application behaves before/after reconfiguration. Model checking techniques provide a formal framework for verification of the dynamic properties of the system [35, 77] and these techniques have been specialized for verifying reconfiguration requirements. With the proposed specialized techniques, one either has to remodel the system in the language of the model checker [60, 76] or one has to model application-specific operational semantics [19, 15, 25]. Thus, in all approaches the verification requires additional artifacts (models or execution semantics) from the users. From these approaches, the references [60, 15, 19] provide verification on UML models where the UML meta-model elements are used for modeling component based architectures and formal execution semantics are provided for these models. However, the proposed semantics are not suitable for verification when the detailed OO design is completed. The verification on the UML models of OO designs should focus on objects and the interactions of objects which are

changed by the reconfiguration. This requires execution semantics that are very close to the actual execution of the OO software.

In this chapter, we describe our approach for providing automated verification of the changes in the interactions between the objects/classes of the application with respect to runtime reconfiguration requirements on detailed designs of OO software systems specified by UML class and sequence diagrams. We only focus on verifying these requirements on the application and assume that the configuration system includes the means for detecting the environmental changes (usually, this part of the configuration system is reused, making it important to study the effects of the reconfiguration on the application). We use graph-based model checking to provide the verification because UML models can be easily represented as graphs and, in our approach, we provide tools for automatically converting UML class and sequence diagrams to graph models; thus, it is sufficient to model the design of the application in UML class and sequence diagrams to use our approach.

The inputs to our approach are the UML class and sequence diagrams of the software system and the execution sequence that conforms to the reconfiguration requirement. For expressing the execution sequence, the users can use Computational Tree Logic (CTL) or a visual state-based language (VSL) developed by us. The verification is realized by simulating the input UML diagrams, which generates a state-space showing all execution sequences supported by the input models. A verification algorithm evaluates whether the generated state-space contains a branch that follows the execution sequence conforming to the reconfiguration requirement. In case such an execution sequence cannot be found, than a feedback mechanism provides guidelines about the location of the problem. Because reconfiguration happens at the operational phase of the software, these diagrams should be simulated with execution semantics that are close to the actual execution of the OO software system. We modeled the execution semantics for UML models using graph transformations and specialized a graph-based model-checking environment called GROOVE [111]. These semantics are generic (i.e. not application-specific) and mimic the actual execution of OO software systems. Graph transformation based generic behavioral semantics modeling of UML models is presented in the literature [85], [86]. However, these semantics are not suitable for verifying reconfiguration requirements, because they do not provide a simulation that is similar to the actual execution of the software system. In summary, the novel contributions of this chapter are:

- A tool for automatically converting UML class and sequence diagrams to a graph-based representation.
- Expressing reconfiguration requirements using CTL [34] and a visual state-based language.

- Implementation of reconfiguration/execution semantics close to OO execution for UML models through graph transformations.
- Automated verification of UML models with respect to reconfiguration requirements.
- Two feedback mechanisms that provide guidelines on the possible locations of the problems when the verification of a requirements fails.

We carried out an extensive case study to test the validity and applicability of our approach.

The rest of this chapter is organized as follows: The next section provides an overall view on the approach. Section 3.2 describes the reconfiguration mechanisms: how these mechanisms are represented in UML and the semantics of these mechanisms. The state-space generated from the simulation is detailed in Section 3.3. Section 3.4 describes how the reconfiguration requirements can be expressed in CTL. The visual state-base language is explained in Section 3.5. The feedback mechanisms that provide guidelines on the location of the problem when verification fails is presented in Section 3.6. The literature that is related to our work is presented in Section 3.7. Finally, Section 3.8 presents the conclusions.

3.1 Graph-based model checking of runtime reconfiguration requirements

Oreizy et al. [107] define runtime reconfiguration as the ability to recombine existing functionality to modify overall software system behavior at runtime. Furthermore, this study provides a component/connector based architectural model where the reconfiguration is performed by altering the connections between components. We follow a similar model in our approach, where we focus on verification of the alteration between components of the application. We assume that a configuration system that can detect changes in the environment is present. When a change in the environment is detected, this configuration system reconfigures the functionality of the application through programming structures, which we call the reconfiguration mechanisms. An example of this model is a configuration system based on configuration scripts. At runtime, the configuration system can parse the configuration script and instruct the components to execute certain blocks by means of conditional statements which test the values read from the script. Here, the conditional blocks are the reconfiguration mechanisms.

The UML models[1] of the application, usually, include the reconfiguration mechanisms that will be implemented in the application to show, for example, where the reconfiguration happens. This provides a sufficient framework for manually tracing the diagrams and verifying that an invariant of the application is not violated or that the application can be correctly reconfigured. However, as presented in the industrial case study (Section 4.1), by manually tracing UML models it is not always possible to know all execution sequences these models support. Due to wrong evaluation, bugs may be introduced to the source code (i.e. the actual software); hence, the benefits of early evaluation are reduced. We apply graph-based model-checking to generate all possible execution sequences supported by the UML models of the application and verify that the reconfiguration happens correctly.

The scope of our approach is non-distributed object-oriented systems, so the bindings between components (classes, methods, etc.) are the calls. The reconfiguration mechanisms describe how the configuration system is able to change the call relations. Given the UML models of the application with reconfiguration mechanisms, our aim is to verify that the interactions can be modified correctly with respect to reconfiguration requirements. To achieve this aim, we simulate the execution and reconfiguration of the UML models (i.e. the sequence diagrams) and generate all possible execution sequences the models support. In this way, for example, we can combine two sequence diagrams and generate an execution sequence that is not modeled. Then, we can reason whether this execution violates an invariant of the system or whether it is a desired reconfiguration. As a result, more bugs related to reconfiguration can be discovered before the implementation.

To simulate UML models and reason about the reconfiguration of the software system, we applied graph-based model-checking. In graph-based model-checking, a runtime state of a system is modeled as a graph, and its behavior is modeled as graph transformation rules (the reader is referred to the literature [77] for more formal and detailed definitions). The graph production tool automatically applies all the predefined transformation rules that match the current graph. This may result in one or more graphs, representing the different states of the system. In this way, the graph production tool simulates the behavior of the modeled system. The simulation generates a state-space (with transitions) showing the possible states the system can reach. In our case, each branch in this state-space is a possible execution sequence of the UML model.

Because reconfiguration happens at runtime and in order to fully capture the effects of reconfiguration on the execution sequence, the reconfiguration and execution semantics modeled as graph transformation rules are very close to actual object-oriented system execution. We modeled the semantics of 4 reconfiguration

[1]In the rest of the chapter, we refer to UML class and sequence diagrams only as UML models.

mechanisms and evaluated our approach with an industrial software that uses these mechanisms. However, the approach is not limited to these 4 reconfiguration mechanisms. It is possible to extend the approach by modeling the semantics of other reconfiguration mechanisms. Once these semantics are modeled as graph transformation rules, they are placed in the same directory as the rules modeling the execution semantics; in this way the simulator automatically triggers these rules as well.

Reconfiguration requirements, usually, describe the changes in the interactions between software components textually. These requirements describe two types of changes in the interactions that need to be verified when designing a reconfigurable software system: 1) a change in the interactions that the application should support and/or 2) a change in the interactions that is not supported by the application and, thus, should be avoided. We refer to the first type of requirements as *supported reconfiguration* and the second type as *reconfiguration invariant*. Because the simulation generates all possible execution sequences supported by the UML models, one needs to express the textual reconfiguration requirements as an execution sequence in order to verify a reconfiguration requirement. For example, one needs to specify the execution order of certain methods to see whether the reconfiguration mechanisms correctly change the interactions for a supported reconfiguration requirement. These execution sequences can be expressed as temporal logic formulas or using a visual language called VSL developed by us. If CTL is used, then the propositions of the formulas used in our approach contain the names of the methods/classes and/or the reconfiguration mechanisms executed. After specifying the execution sequence, a verification algorithm searches the state-space to find the states that satisfy the temporal logic formula.

In general, the graph-based model-checking for evaluation of runtime requirements is performed as follows:

1. The designer provides the class diagram and the corresponding sequence diagrams of the design, using UML notation. The sequence diagrams include special tags that describe the reconfiguration mechanisms used in the design.

2. The class diagram and the sequence diagrams are converted into DCML (detailed in Section 2.1).

3. The execution of the UML models, in DCML representation, is simulated using graph transformation rules modeling the execution/reconfiguration semantics of the UML models. The simulation generates the runtime reconfiguration state-space of the model. The GROOVE tool is used for the simulation and the generation of the state-space.

4. The Requirement Engineer specifies the runtime reconfiguration requirements and expresses the execution sequences conforming to these requirements using CTL formulas or using VSL. In case the requirements are specified using VSL, they are converted to CTL formulas with a converter we developed. These formulas are verified by the model-checker implemented in GROOVE. This verification marks the states that satisfy the CTL formula in the state-space.

3.1.1 Computational Tree Logic

Computational tree logic (CTL) is a branching time logic whose model is a tree-like structure [34]. CTL has logic operators, such as *and* ($\wedge$) and *or* ($\vee$) and temporal operators. There are two types of temporal operators in CTL: quantifiers over all paths (i.e. branches of the tree) and quantifiers specific to a path (i.e. a branch of the tree). Below these quantifiers are described (assume x and y are properties or state names):

1. Quantifiers over paths:
   - *A(x)*: There is a state (node) in the tree where starting from this state x holds on all paths.
   - *E(x)*: There is a state in the tree where starting from this state x holds on at least one path.
2. Quantifiers specific to a path:
   - *F(x)*: is an path specific operator and means that eventually in the subsequent path x has to hold.
   - *G(x)*: in the entire subsequent path x has to hold.
   - *N(x)*: at the next state x has to hold.
   - *(x U y)*: x has to hold until at some state y holds.

In CTL the path operator must be followed by a path specific operator. For example $EF(x)$ means there exists a path in the tree where somewhere in this path x holds. A CTL formulae is generated by the following syntax: $x := p|\neg x|x \wedge x|x \vee x|x \Rightarrow x|x \Leftrightarrow x|AX(x)|AF(x)|AG(x)|A(xUx)|AN(x)|EF(x)|EG(x)|EN(x)|E(xUx)$, where p is an element of the set of atomic propositions.

The graph transition system (GTS, see Section 2.2) of a graph production system can be mapped to the tree-like structure that CTL operates on (Kastenberg et al. [77] provide this mapping). In this mapping the names of the graph transformation

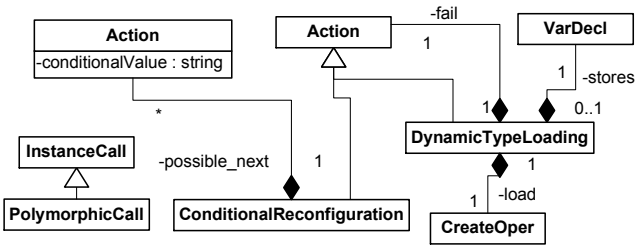


Figure 3.1: The meta-model showing the graph elements used for declaring the reconfiguration mechanisms in the DCML.

rules (which are also used for labeling the edges in the transition system) become the names of the states.

3.2 Reconfiguration Mechanisms

The reconfiguration of object-oriented systems causes changes in the calls between software entities. The implementation specific details on how this reconfiguration happens may not be captured in sequence diagrams (it may be too detailed). However, the designer can specify the *reconfiguration mechanism* the software will support in the UML models. A reconfiguration mechanism captures the semantics of the reconfiguration; that is, how the calls between software entities are changed.

Similar to the execution simulation semantics (Section 2.2), the reconfiguration semantics are also modeled using graph transformation rules and the simulation of these semantics occurs automatically by triggering the appropriate graph transformation rules. These rules are triggered when the execution reaches an action that is reconfigurable.

A reconfigurable action specifies that after this action, the execution has some alternative paths. The reconfiguration transformations generally select one of these alternative paths. State-space generation continues until none of the rules match; so, at the end of the simulation each alternative path has been taken and its execution has been simulated.

We modeled the semantics of 3 kinds of reconfiguration mechanisms: polymorphic reconfiguration, conditional statements and dynamic type loading. The reason for selecting these reconfiguration mechanisms is twofold:

1. These mechanisms are supported by most of the object-oriented programming languages. Because UML models can be implemented in any OO language,

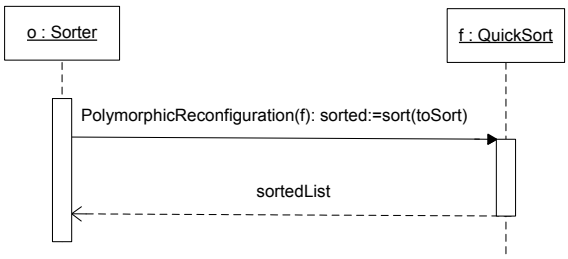


Figure 3.2: The call action *sort()* tagged with polymorphic reconfiguration.

we wanted to add mechanisms that can be used in many OO languages.

2. An elaborate study on the MRI software system showed that these mechanisms are indeed used often to reconfigure the software.

Nevertheless, the approach is not limited to these mechanisms. We designed the approach such that the reconfiguration and execution semantics are distinct; when the simulation reaches a reconfigurable action, first the reconfiguration of it is simulated and then the execution semantics continue the simulation from the reconfigured action. A new reconfiguration mechanism can be introduced by extending DCML with elements required for representing the new reconfiguration mechanism (i.e. adding a new sub-class for the node *Action*) and modeling the semantics of the reconfiguration mechanism with graph transformations (In reference [30], we shown an extension of the approach).

Figure 3.1 shows how the reconfigurable actions for these 3 mechanisms are represented in DCML. The designer specifies the reconfiguration mechanism to be used and the actions affected by the reconfiguration in the UML sequence diagrams. We added tags to the UML sequence diagram actions/guards for this purpose; the tags are entered next to the name of the action in ArgoUML. Below the modeled reconfiguration mechanisms and how they are specified in UML are described:

1. **Polymorphic Reconfiguration**: In a UML sequence diagram the call action designating a polymorphic reconfiguration should be tagged as *[PolymorphicReconfiguration(reference_variable_ name)]*. With this reconfiguration mechanism, the designer specifies that a call action can be received by any object-type that is a sub-type of the type of the reference variable. The UML tag for the call action *sort()* is presented in Figure 3.2. In DCML a call node that is reconfigurable using polymorphism is represented by nodes labeled *PolymorphicCall* (Figure 3.1). The transformation rule modeling this reconfiguration mechanism is presented in Figure 3.3. For this rule to match, the simulation should be executing an instance call action that is also labeled

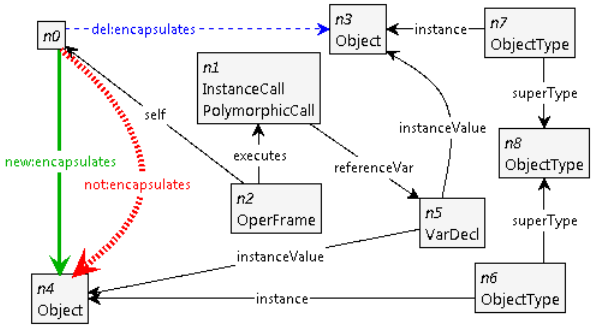


Figure 3.3: The graph transformation rule modeling polymorphic configuration.

as *PolymorphicCall*. The rule changes the object the reference variable of a call holds to one of the objects that are type-compatible with the type of the reference variable. As a result, each type-compatible instance the call during simulation. The transformation rules in Figure 2.14-(a) and in Figure 3.3 have the same priority; so, the call can either execute without reconfiguration or polymorphic reconfiguration happens.

2. **Conditional Reconfiguration**: Conditional statements such as *if-else* and *switch-case* can be used to reconfigure the software system at runtime. In UML 2.0 sequence diagrams, conditional execution of the actions is represented by *frames* whose operators are *alt* or *opt* [55] (*alt* stands for alternative and *opt* stands for optional). A frame can have one or more fragments which contain the actions. A fragment with a guard shows that the actions within the fragment are executed when the guard is true (the ArgoUML support for UML 2.0 style conditional statements is under development). Figure 3.4-(a) depicts a sequence diagram with two frame fragments, which are in the scope of the method *Main.Create()*. Depending on the value of the attribute *init*, one of the fragments is executed: if *init=true* then the fragment that calls the method *Program.getProgram()* is executed and if *init=false* the fragment that returns *null* is executed.

   As can be seen from Figure 3.3, conditional frames are treated as actions with multiple *next*'s in DCML. Figure 3.4-(b) represents this sequence diagram in DCML. Here, the emphasized node labeled *ConditionalReconfiguration* represents the frame (conditional reconfiguration node). In scope of the method *Main.Create()*, the first action is a frame; thus, in the DCM of this sequence diagram the first statement of this method is a conditional reconfiguration node. The fragments of a frame are represented by edges labeled *possible_next*, connecting the conditional reconfiguration node to the first action node of the fragment. The node that is the first action of a fragment also gets an attribute

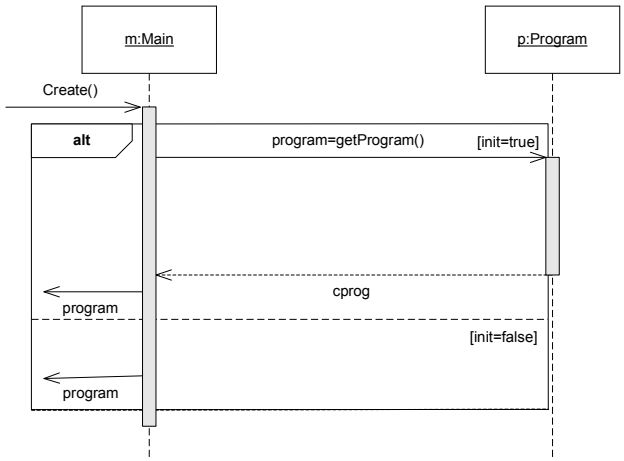
named *conditionalValue*, which represents the guard of the fragment. For example, the method call node with the attribute *conditionalValue* set to *init = true* (Figure 3.4-(b)) represents the first call action of the fragment with guard *init = true* in the sequence diagram of Figure 3.4-(a).

Figure 3.5 depicts the graph transformation rule modeling the semantics of conditional reconfiguration. When the execution reaches a node labeled *ConditionalReconfiguration* then this rule matches. A match of the rule converts one of the alternative paths to a direct path by replacing the edge labeled *possible_next* by an edge labeled *next*. For an alternative frame with n fragments (i.e. n edges labeled *possible_next*), this rule has n matches.

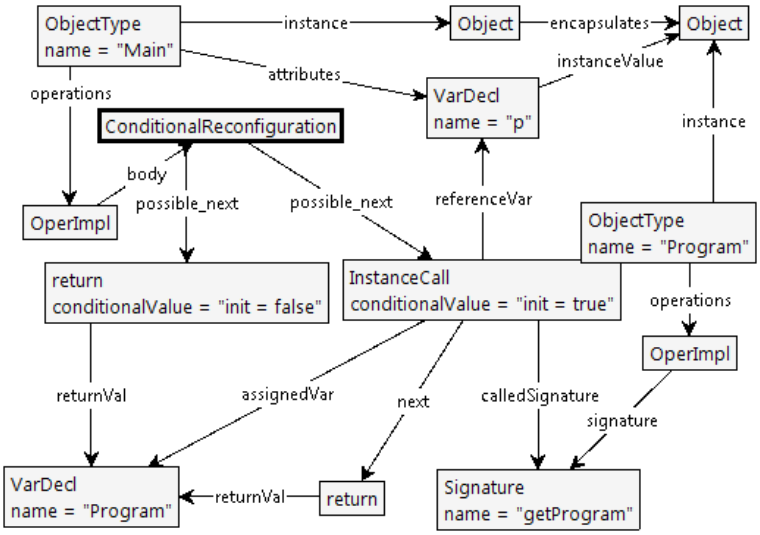
3. **Dynamic Type Loading**: this is the reconfiguration mechanism where a class is loaded at runtime. In UML sequence diagrams, this mechanism can be represented by alternative frames, where one fragment shows the actions taken when the class is successfully loaded and another fragment shows the actions taken when the loading fails. The fragment that shows the successful load of the type has the guard *DynamicTypeLoading(className)* where *className* is the name of the class to be loaded. The fragment showing the actions taken when the load fails, on the other hand, has the guard *DynamicTypeLoading_fail(className)*. Usually, when a class is successfully loaded, one of its constructors is called. Figure 3.6 depicts a sequence diagram with dynamic type loading. Here, the class *Program* is dynamically loaded. When this class is successfully loaded an instance of it is created and stored in the attribute named *p*. When the class loading fails, *null* is returned by the method *Main.loadProgram()*.

   A frame with a fragment whose guard is dynamic type loading is represented in DCML by an action node labeled *DynamicTypeLoading*. In the sequence diagram of Figure 3.6-(a), the first action at the control scope of the method *Main.loadProgram()* is an alternative frame with fragment guards that use dynamic type loading. In the DCML representation of this sequence diagram, Figure 3.6-(b), the emphasized node is the fragment from which the type is loaded. The create action that is executed after the type is loaded is shown with the edge labeled *load* connecting the dynamic type loading fragment node to the node labeled *CreateOper*. The failure fragment is shown with an edge connecting the dynamic type loading node to an action node labeled *fail*.

   The semantics of the dynamic type loading is captured with three graph transformations. The transformation rule presented in Figure 3.7-(a) models the semantics of the successful load of the type. For this rule to match, the class to be loaded should be in the DCM (i.e. it should be specified in UML diagrams). In the transformation rule, the node n represents the object-type to be loaded.



(a)



(b)

Figure 3.4: a) Example sequence diagram with conditional frame fragments b) The sequence diagram represented in DCML

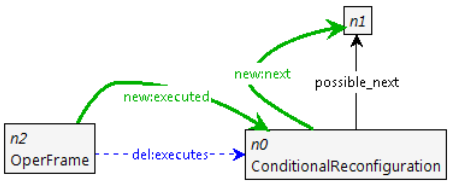


Figure 3.5: The graph transformation rule modeling conditional reconfiguration.

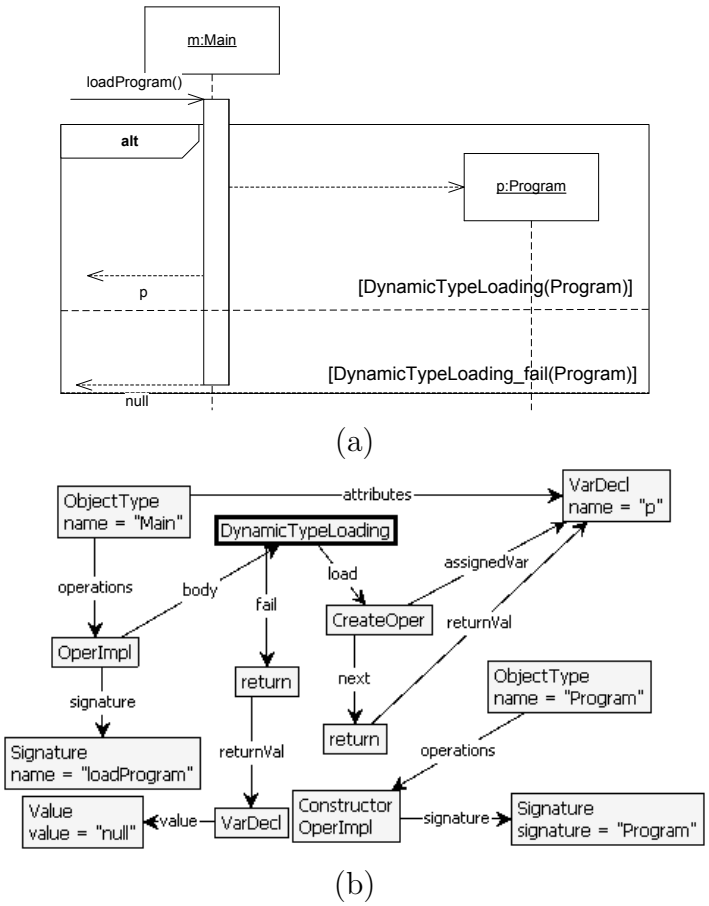


Figure 3.6: a) An example UML sequence diagram showing how dynamic type loading is modeled b) The DCM of this sequence diagram.

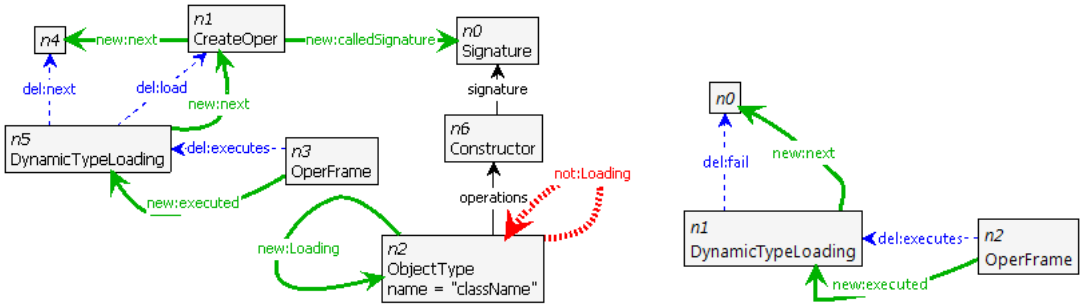


Figure 3.7: a) The graph transformation rule that selects the fragment containing the actions when the type is successfully loaded. b) The graph transformation rule that selects the failure fragment.

Because the name of this object-type is specified by the user, this rule is *generated* according to the name supplied in the sequence diagram. For example, the loading of the class *Program* is modeled in the sequence diagram with a frame fragment that has the guard *DynamicTypeLoading(Program)* (Figure 3.7-(a)). During conversion, the UML-to-DCML converter creates a copy of the transformation rule presented in Figure 3.7-(a) and replaces the name *className* with *Program* (note that the rule also is renamed so that more then one type can be loaded). When the execution reaches a dynamic type loading node (i.e. a node labeled *DynamicTypeLoading*) and if all the requirements of this rule are satisfied, then the rule matches. The application of this rule replaces the edge labeled *load* by an edge labeled *next*. This causes the simulation to execute the *CreateOper* action after the dynamic type loading action. That is, an instance of the newly loaded type can be created when the operation frame moves to the next action. The rule also adds an edge labeled *calledSignature* from the *CreateOper* action to the signature of the constructor of the loaded type.

The transformation rule in Figure 3.7-(b) also matches when the execution reaches a dynamic type loading node. However, this rule replaces the edge labeled *fail* by an edge labeled *next*. Thus, this rule selects the fragments containing the actions that are executed when loading fails.

It is possible in UML to model dynamic type loading with optional frame fragments or with alternative fragments that do not include the fail guard. That is, it is possible not to specify the actions executed after the dynamic type loading fails. In the DCMs of such sequence diagrams, the edge labeled *fail* does not exist. In such diagrams, the rules presented in Figure 3.7-(a) and Figure 3.8 match. The latter rule adds an action node labeled *exception* as

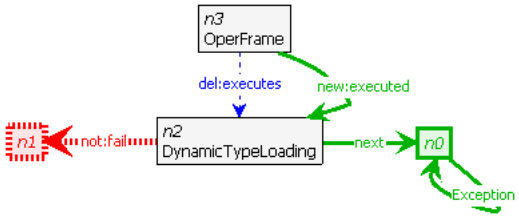


Figure 3.8: The rule that simulates type loading exception.

the next action of the dynamic type loading. The added node does not have any next actions; thus, when the execution reaches that action, it stops. This represents the case that the exceptions thrown by dynamic type loading are not handled by the program. Note that these rules have the same priority and when the simulation reaches the dynamic type loading node, then it can either select the failure path or the successful load path.

3.3 The transition system resulting from simulation

To remind the reader, the simulation of the sequence diagrams generates a state-space (with the transitions) called a *graph transition system* (GTS) (for a more detailed description the reader is referred to the literature [77]). In a GTS, the labels of the nodes are the names of the states and the labels of the edges are the names of the graph transformation rules that are applied. Thus, from the GTS we can observe the methods/objects that have received calls.

Figure 3.9 depicts the UML models for a sorter system that follows the strategy pattern [59]. The sequence diagram on the bottom shows that the call action *sort* is polymorphic; the receiver of the call can be changed at run-time. In Figure 3.10, the important states of the graph transition system resulting from the simulation of the sorter system (Figure 3.9) are presented.

In the transition system generated from the execution and the reconfiguration simulation of a DCM, a path from the start state (state labeled as *start*) to a final state (state labeled as *final*) is an execution sequence. In the GTS resulting from the simulation of the sorter software (Figure 3.10) there are two execution sequences; after state *s10* the execution continues in two branches. The label *Polymorphic-Configuration* of the transition between states *s10* and *s12* shows that at state *s12* the transformation rule that models the polymorphic reconfiguration is applied (we

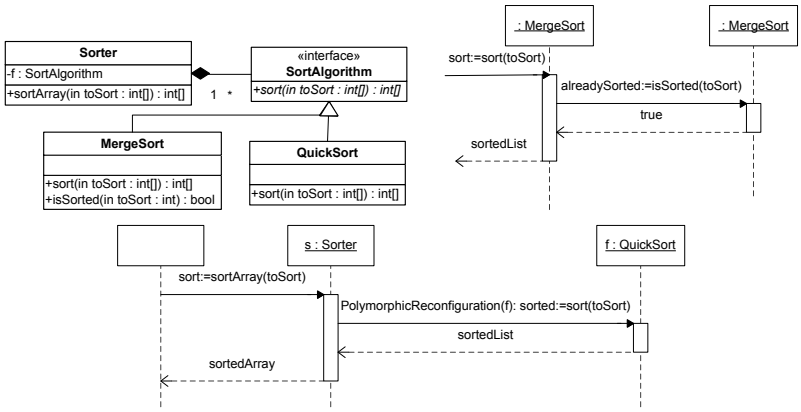


Figure 3.9: The UML models of a sorter system that uses polymorphism to switch between sort algorithms

presented this rule in Figure 3.3). Since the software entities required by this reconfiguration are in the design of the sorter system, this rule matched to the design and changed the value of the variable *f* to an instance of the class *MergeSort*. Although it is not specified in any of the sequence diagrams, an execution sequence showing the method *MergeSort.sort* executing after the call to *Sorter.sort* is generated. The simulation was able to combine the two sequence diagrams because the call is specified to be reconfigurable using polymorphism. It is important to note that the application of every reconfiguration transformation to a DCM can generate a new execution sequence.

There are three important transitions that give information about the methods, together with the instances of types that executed during simulation of a DCM; these are the transitions with the labels *executes(object-type name)*, *executeMethod(method name, object-type name)* and *returnframe(method name, object-type name)*; these are special graph transformation rules that use GROOVE's built-in mechanisms to display the values of the attributes of the nodes. For example, in the path that ends with state *s37* in Figure 3.10, there is a transition with the name *executes("quickSort")*; so, in this path the call *f.sort* is received by an instance of the object-type *quickSort*. Figure 3.11 presents the transformation rule *executeMethod*. This rule matches when the operation frame starts executing a method. The edge labeled *executes* from a operation frame node points to an action node; however, before the method starts executing it points to the operation implementation node. When this rule is applied, it moves the edge labeled *executes* to the first action of the method body. In this way, the point where the method starts executing is captured. The nodes $x64$ and $x63$ are used to display the name attribute of the signature node and the object-type node respectively. These names are displayed with the transformation rule name in the generated state space as shown in Figure 3.10.

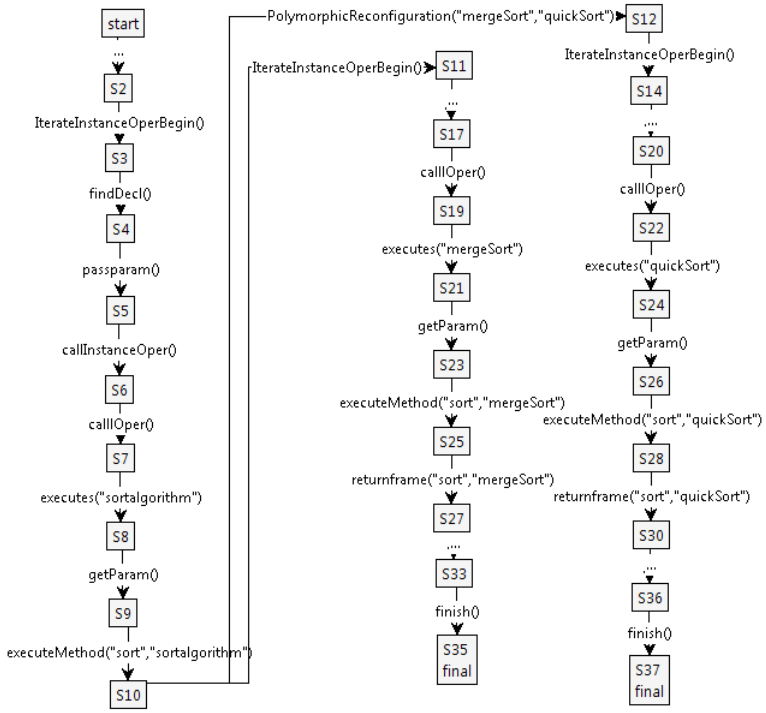


Figure 3.10: The GTS resulting from the simulation of the design presented in Figure 2.2.

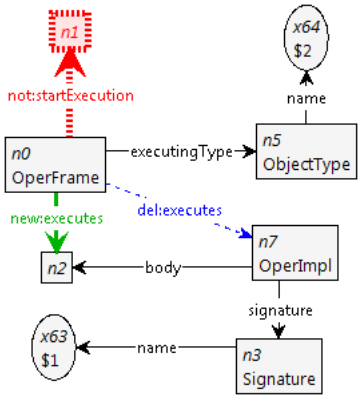


Figure 3.11: The transformation rule used for getting the name of the method that started executing.

The transition system also includes the names of the applied reconfiguration mechanisms such as *PolymorphicReconfiguration*, *DynamicTypeLoading_fail*. Thus, the GTS can be searched for the names of the reconfiguration mechanisms to learn where the reconfiguration mechanisms are applied during simulation.

3.4 Expressing Reconfiguration Requirements in CTL

The GTS resulting from the simulation shows all the execution sequences the UML model supports. Thus, to verify that a sequence is supported, one should find a state (or states) in the GTS from which the objects/methods execute in the provided sequence. This can be formalized and automated by expressing the execution sequence of the requirement as a CTL formula and running the CTL verification algorithm on the GTS (this algorithm is implemented in GROOVE).

In section 3.3, the transition labels that give information about the executed methods/objects and the reconfigurations mechanisms during the simulation were described. Using the same labels and sequencing these labels with the operators of CTL, an execution sequence can be expressed in CTL. For the sorter system example, we can formalize the runtime reconfiguration where the implementation of sort is changed to *quickSort* with CTL as follows:

$$EF(executeMethod(\text{"}sort\text{"},\text{"}Sorter\text{"}) \wedge (EF(executes(\text{"}quickSort\text{"})) \\ \wedge(EF(returnframe(\text{"}sort\text{"}.\text{"}Sorter\text{"})))))$$

This formula looks for states from which the method *Sorter.sort()* starts executing, then an instance of class *quickSort* starts executing and, lastly, the method *Sorter.sort* returns.

3.5 Visual State-Based Configuration Specification Language

Case studies conducted with the industry showed that CTL was hard to use by the designers (see section 4.1). As a result, we developed a visual state-based configuration language called VSL for expressing the execution sequence that conforms to

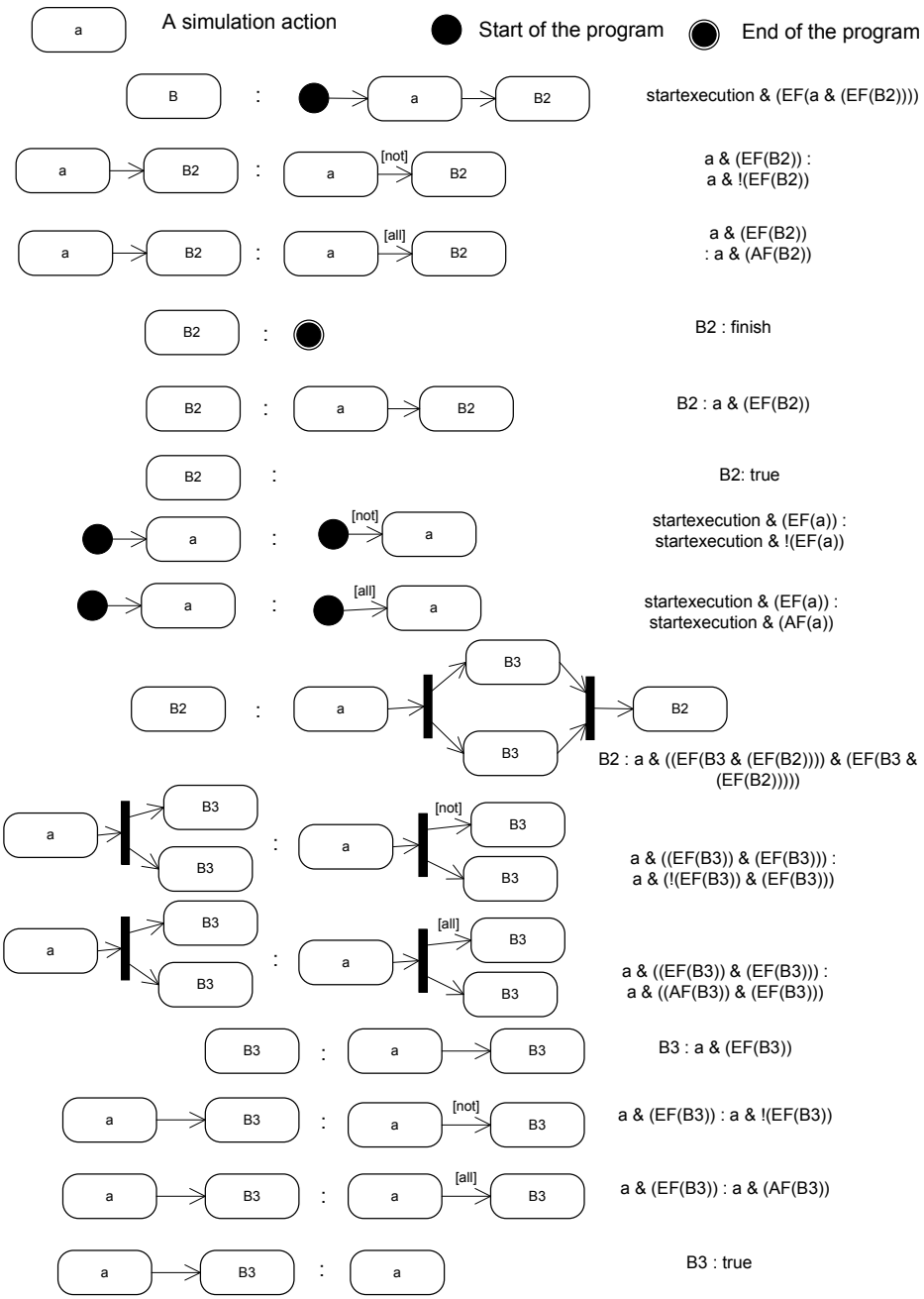


Figure 3.12: The grammar of VSL with the conversion from VSL to CTL.

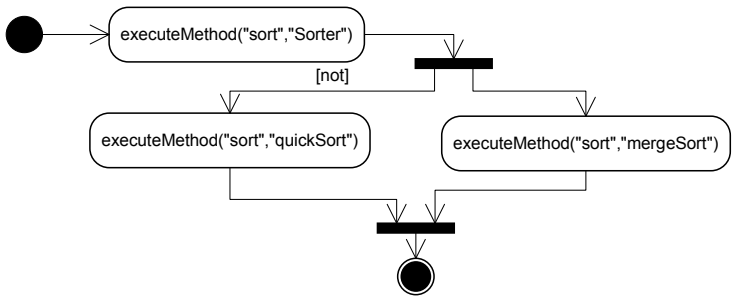


Figure 3.13: A VSL specification that looks for an execution where an instance of the class *mergeSort* receives the call *sort* and *quickSort.sort* does not execute.

the reconfiguration requirement. VSL uses UML state/activity diagram elements and covers a subset of CTL that is used to express the requirements for the case studies. This subset includes $EF, AF, !, \bigwedge$. We have extended GROOVE with a translator for translating a VSL specification into a CTL formula and with the *VSL verification* panel used for entering a VSL specification. It is important to note that complex execution sequences (i.e. execution sequences that require other CTL operators) can still be entered using GROOVE's CTL formula editor.

Figure 3.12 represents the grammar of the VSL. Here, the states are used for representing a simulation action (i.e. the atomic propositions) like *executeMethod*. The initial and final states correspond to *startexecution* and *finish* simulation actions respectively. The transitions represent the order of the simulation actions; the CTL equivalent of a transition is $\wedge(TemporalOperator(x))$, where $TemporalOperator$ can be EF,AF,$!EF$ depending on the guard of the transition (the guard can be *none*, *not* and *all*). Note that x in the previous formula is the state that is connected to the arrow head of the transition.

It may be required to reason about two things after a simulation action; for example, after *a*, *b* should execute and after *a*, *c* should execute. These sequences can be expressed in VSL using the fork transition. Figure 3.13 specifies the execution sequence where after the method *Sorter.sort()* begins its execution eventually the method *mergeSort.sort()* executes but *quickSort.sort()* does not execute. This specification is translated into the following CTL formula:

$$\begin{aligned} &startexecution() \wedge (EF(executeMethod(\text{"}sort\text{"}.\text{"}Sorter\text{"}) \\ &\wedge(EF(executeMethod(\text{"}sort\text{"}.\text{"}mergeSort\text{"}) \wedge (EF(finish())))) \\ &\wedge!(EF(executeMethod(\text{"}sort\text{"}.\text{"}quickSort\text{"}) \wedge (EF(finish()))))) \end{aligned}$$

It is important to report here that we did not specifically ask the designers to express

an execution sequence conforming to the reconfiguration requirement in VSL after designing it. We conducted two experiments to test our approach and in these experiments VSL are used. These experiments were conducted with students (as we detail in Chapter 4) and we wanted to use requirements that are close to actual reconfiguration requirements of the industrial software we used during the case study with the designers. Thus, we took some simpler requirements of this industrial tool, changed them and expressed them using VSL. The designers saw these sequences and they were able to grasp what they mean. More importantly, the students were able to use VSL after a 10 minute explanation. This shows that VSL is indeed easier to use and we think that the major reason for this is because VSL uses elements that most computer scientist are familiar with.

3.6 Providing Error Diagnosis

If the VSL specification is supported by the UML models, then the number of counterexamples (the states that do not satisfy the formula) is less then the number of states the simulation generated (this is because any state after *startexecution* does not satisfy the formula). If, on the other hand, the formula does not hold then the number of counterexamples is equal to the number of states generated by the simulation. Using this information, a text message whether the VSL specification is supported by the UML models or not is presented in the VSL verification pane. However, in case the requirement is not supported, more insight on the possible locations of the requirement failure is more helpful for the designers.

As discussed before, there are two types of reconfiguration requirements: 1) supported reconfigurations 2) reconfiguration invariants. The evaluation of a CTL formula for these requirements displays the states from which the execution sequence we are searching for can be reached. This makes it hard to trace the source of the reconfiguration problem. For example, after a state there may be too many branches, making it hard to manually trace the problematic execution sequence. Because of this, we developed two error diagnosis mechanisms. The first one is a CTL formula tracer that tries to find the state where the CTL formula evaluates to false. This tracer can only be used for the first type of requirements.

The second error diagnosis mechanism is based on control automata. In this mechanism, the VSL specification is converted into a control automaton with accept and reject states. After the simulation, the states from which an accept state can be reached are found. If a requirement is found to be not supported by the UML models, then for the first type of requirements this mechanism outputs the whole execution trace up to the point for which the requirement is supported. For the

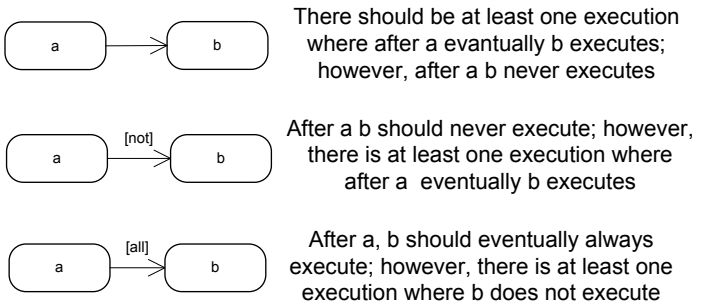


Figure 3.14: The messages that provide insight on the possible location of the problem for different VSL transition types.

second type of requirements, this mechanism does not work.

We extended GROOVE with a verification panel, where the user can enter VSL specifications and automatically get the verification results. Depending on the configuration, the verification panel uses one of the error diagnosis mechanisms. This section details the two error diagnosis mechanisms and provides a discussion on their limitations.

3.6.1 Providing Error Diagnosis with CTL

When the UML models should support an execution sequence (i.e. the first type of reconfiguration requirement) but the simulation does not yield such an execution sequence, then showing the point where the VSL specification has failed provides more insight to the designers. To provide such an insight, we implemented a VSL tracer which removes the last transition and state from the VSL specification and then evaluates the new VSL specification. The tracer repeats removing the last state and transition until the VSL specification evaluates to true; thus, the tracer finds the point up to which the original specification is supported and the tracer prints out this state with what went wrong during evaluation. Figure 3.14 presents the messages printed out by the tracer for different VSL specification types (point of failure is between a and b). Note that if the tracer is left only with the *start* state then it displays the message: "the UML models cannot reach the first simulation action". For VSL specifications that contain forks, the tracer provides a guideline message for each branch that is not supported by the UML models.

In Figure 3.15-(a), an example VSL specification is presented. Assume that the specification evaluates to false; that is, it is not supported by the UML models. The VSL tracer tries a find the state in this specification after which something went wrong and the specification evaluated to false. For this, the tracer removes

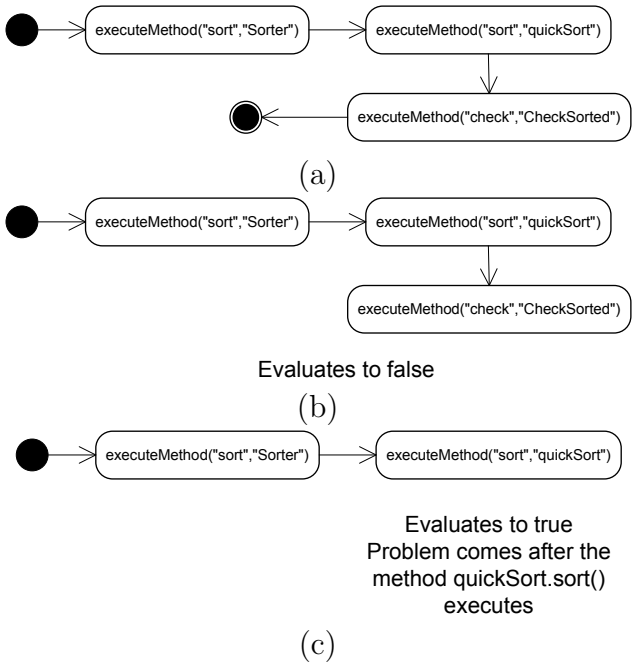


Figure 3.15: a) An example VSL specification that is not supported by the design. b) The VSL tracer moves the last state from the specification and re-evaluates the specification. The evaluation also yields to false. c) Since the specification at (b) evaluates to false, the tracer removes the last state and re-evaluates the specification. This time the evaluation yields to true; the problem is related to execution after *quickSort.sort()*.

the last state, which is the program finish state, resulting in the specification presented in Figure 3.15-(b). The tracer evaluates this new specification. Assume that it also evaluates to false; so, the tracer removes the last state, the state *executeMethod("check","CheckSorted")*, resulting in the specification shown in Figure 3.15-(c). Assume this specification evaluates to true; this means that after the method *QuickSort.sort()* started execution, the method *ChecksSorted.check()* did not execute. The problem with this specification is related to a point after which *QuickSort.sort()* starts executing and designers need to focus on this part. To provide the guideline, the tracer prints: *"There should be at least one execution where after QuickSort.sort() eventually CheckSorted.check() executes; however, after QuickSort.sort() CheckSorted.check() never executes"*.

The major limitation of this error diagnosis mechanism is that it provides a trace based on the specification. That is, it cannot provide the whole execution trace. The verification of an invariant is done by searching for an execution sequence that violates the invariant. If such an execution sequence is found, then the trace should

be presented to the user (i.e. from the *start* state). Since this error diagnosis mechanism cannot provide such an output, it cannot be used to get insight for the violations of the invariants.

3.6.2 Providing Error Diagnosis with a Control Automaton

The major drawback of using CTL for verification of reconfiguration requirements is that due to the semantics of CTL, it is very hard (sometimes impossible) to get a full trace of the execution sequence when the verification fails. With a CTL tracer, we tried to provide insight on what went wrong during the evaluation of the specification; however, one may still need to trace long sequences of calls until discovering the problem. When one focuses on verification of only one property of one branch after a VSL state (i.e. without the VSL *all* transitions and *transition forks*), then the VSL specification looks very similar to a state-machine. The verification with a state-machine is realized in terms of *accept* and *reject*: the machine goes to an accept state when all the simulation actions specified in the VSL specification are seen or it goes to a reject state when a series of actions that we do not want to observe is in fact observed. If we treat these accept and reject states as simulation actions, then the generated state-space also contains them. Thus, providing a guideline for a failed reconfiguration requirement would involve traversing the state-space and outputting all the branches that lead to a reject state.

To include the accept and reject states to the state-space, we modeled transformation rules with the same name that do not modify the underlying DCML model. That is, when these rules match, they only add transitions to the state-space named *accept* or *reject*. The state-machine can easily be expressed in GROOVE's control language. This language allows one to specify a control automaton that superimposes the transformation engine. In other words, the rules are applied according to this automaton rather than free-form application. We extended GROOVE with a VSL to control automaton converter. For a given VSL specification, this converter automatically generates the control program from which GROOVE generates the control automaton. After the DCM is loaded, the simulation follows this control automaton. This is different from the verification based on CTL. To verify a reconfiguration requirement using CTL, the DCM is simulated and the whole state-space is generated. Then, the CTL formula is entered and the verification algorithm verifies the formula over the whole state-space. However, in verification with a control automaton, the simulation generates the state-space according to the specified automaton.

In this subsection we detail how the control automaton is used in runtime reconfiguration requirement verification. First, we briefly introduce the concepts of the

control program that are needed in order to understand the VSL to state machine transformation below before going to the details of the conversion. For a full semantic description of the control language, the readers are referred to the literature [110].

Each transition of the control automaton refers to a transformation rule. Thus, each statement in the control language is a transition and the statements consist of the name of the transformation rule. Below an example control program is presented:

```
1: doAction1;
2: doAction2 | doAction3;
```

According to this program, when the simulation starts, that is at the start state S_{start}, the transformation engine can only apply the rule named *doAction1*. After applying this rule, the simulation moves to the next state S_1. At this state, there are two outgoing transitions; the simulator can apply either one of the rules *doAction2* and *doAction3*. Assume that applying the transformation rule *doAction2* takes the system to the state S_2 and applying the rule *doAction3* takes the system to that state S_3. If on the graph at state S_1 the transformation rule *doAction2* does not match, then the simulation generates a state space in which after S_1 the state S_3 comes. If, however, both rules match in the state S_1, then the generated state-space contains two branches; one ending with state S_2 and the other ending with state S_3.

The control language also includes control flow statements such as conditional and loop statements. For these the *while* loops and *try* statements are used to express the state-machine conforming to a VSL specification. A *try* statement allows the system to move to the next state in the control automaton when the statements within the try block do not match. Assume that the transformation rule *doAction1* does not match for the input graph. Since the simulation can only change state after applying this transformation rule, the simulation halts at the start state. This may be a desired behavior, when the rule *doAction1* is required to match for all graphs. However, if there may be graphs where *doAction1* does not match, then we can make this rule optional by:

```
1: try{
2: doAction1;
3: }
4: doAction2 | doAction3;
```

Here, if the rule *doAction1* can be applied then it is applied when the simulation moves to the next state. If this rule cannot be applied then it is skipped and the transformation engine tries to apply either of the rules *doAction2* and *doAction3*.

It may also be desired to apply a series of transformation rules until another trans-

formation rule does not match. Such a control flow is expressed with the *while* loop. For example, the control program given below continues on applying the rule *doAction2* and, then, *doAction3* until the rule *doAction1* does not match:

```
1: while(doAction1) do{
2: doAction2;
3: doAction3;
4: }
```

One can express an infinite loop as follows:

```
1: while(true) do{
2: doAction1;
3: doAction2;
4: doAction3;
5: }
```

Here, the transformation engine is instructed to apply first the rule *doAction1*, then the rule *doAction2*, then *doAction3*. After applying the rule *doAction3*, the loop returns back to the rule *doAction1*. The simulation for this control program terminates when one of these rules cannot be applied in the order they should be applied.

With control programs, it is also possible to use functions. The function calls are treated as transitions such that the function call happens when the first rule in the function matches. The function returns after the last rule is applied. Below is an example control program with a function:

```
1: doAction1;
2: foo() | doAction3;
3: function foo(){
4: doAction2;
5: doAction3;
6: }
```

Here, at line 2 the control program states that the transformation engine can either apply the rule *doAction2* or the rule *doAction3*. After applying the rule *doAction2* the transformation rule can only apply the rule *doAction3*.

State-VSL

As discussed before, the VSL *all* transitions and transition forks cannot be used when the control automaton based feedback mechanism is used. The *not* transition

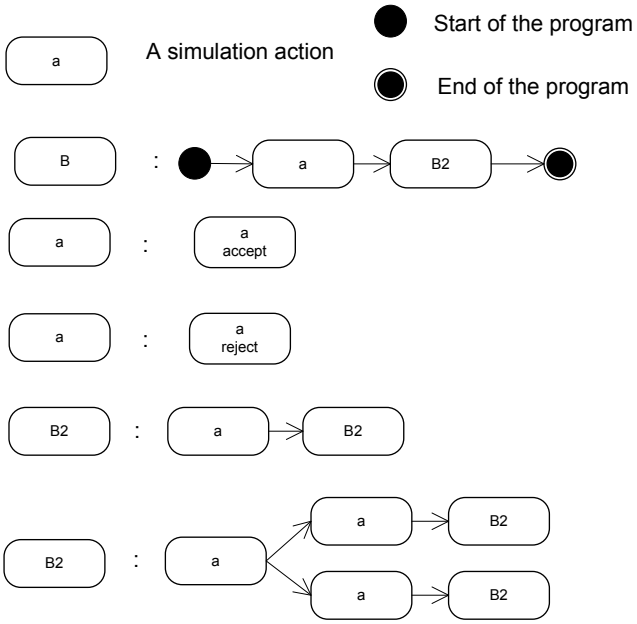


Figure 3.16: The grammar of State-VSL

can be achieved with a reject state; that is, the last state in an execution sequence that we do not want to observe is marked as reject. Due to these differences and limitations, the control automaton based feedback mechanism accepts the input specifications to be expressed in *State-VSL*.

State-VSL is a variant of VSL that uses the same elements (e.g. states and transitions) without the different transition types; Figure 3.16 shows the grammar of State-VSL. The states specify the simulation action that should be observed and the transitions order these actions. The temporal property *eventually* of the transitions also hold in State-VSL. The accept and the reject states are represented by labeling the transition also *accept* and/or *reject*. Note that a State-VSL specification always ends in a *end of the program*.

Although State-VSL does not support transition forks, a state can have more than one outgoing transition, as shown in the last rule of the grammar in Figure 3.16. A state with more than one outgoing transition allows one to reason about two or more distinct branches after a state. In this way, it is possible to provide more details about the branch one is searching. For example, if we are interested in verifying a branch where after the method a eventually the method b executes without executing before the method c, then we can express this in State-VSL with two transitions after a where one goes to a c after a and, then, to the reject state b and the other one goes to the accept state b after a.

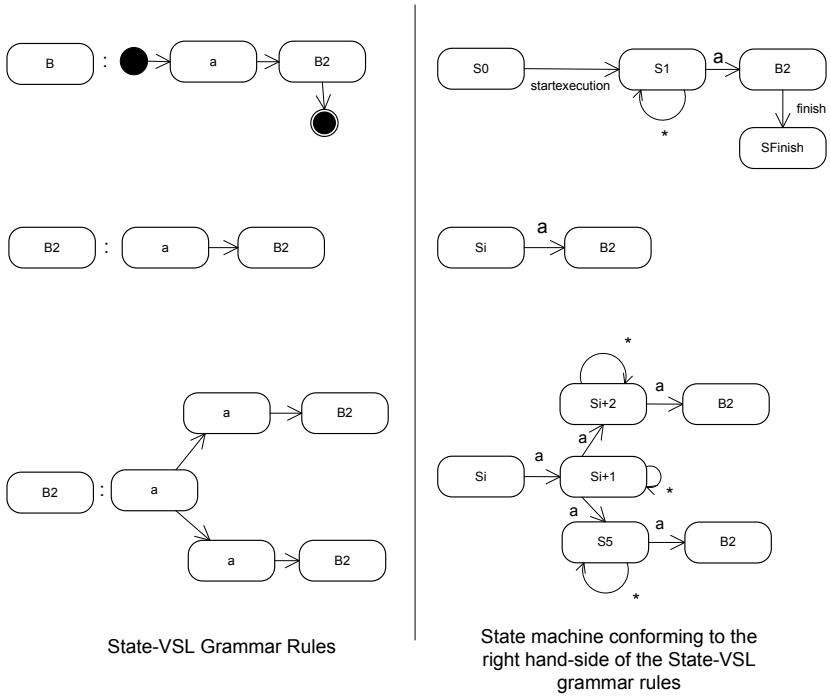


Figure 3.17: State-VSL Grammar Rules and conversion of these rules to state-machines

Conversion from VSL to State-Machines

The semantics of a State-VSL specification can be expressed as a state machine, where the machine is only allowed to change state when a simulation action specified in a State-VSL state is observed. This can be achieved by converting a State-VSL transition VT_i to a state S_i with a wildcard self transition. The wildcard self transition matches to any simulation action. A State-VSL VS_k with label L that is on the end of the transition VT_i is converted to an outgoing transition T_k with label L from the state S_i. The VSL state representing the beginning of the program is converted to a state S_0 with the outgoing transition *startexecution*; here, *startexecution* is the simulation action that prepares the execution of the first call in a DCM. The VSL state representing the end of a program is converted to a state S_{final} with one incoming transition labeled *finish*. The simulation action *finish* happens only after the last call of the DCM is simulated (i.e. the last call action in the input sequence diagram). Figure 3.17 shows the conversion of the State-VSL grammar rules to the state-machines; the conversions of *accept* and *reject* states are omitted as they are explained later in this section.

Figure 3.18-(a) represents an example State-VSL specification where after the pro-

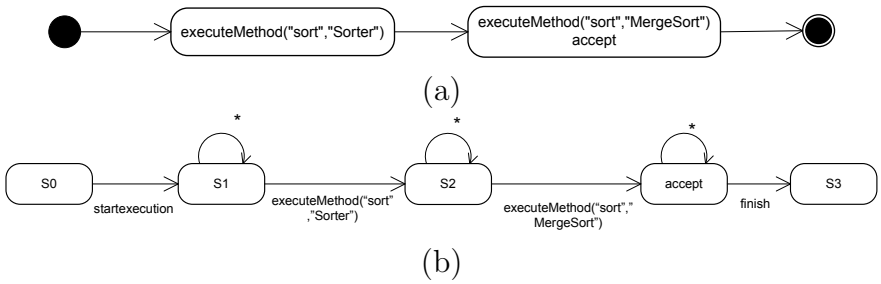


Figure 3.18: A State-VSL specification (a), its equivalent state machine (b).

gram starts execution eventually the method *Sorter.sort()* executes and after this, eventually the method *mergeSort.sort()* executes. The state machine that expresses this VSL specification is presented in Figure 3.18-(b). Here, the transition * is the wildcard transition. Thus, the state machine stays in the state *Start* until the simulation starts executing the method *Sorter.sort*. When this method starts executing the transformation rule *executeMethod* matches and the simulation action *executeMethod("sort","Sorter")* is observed; so the transition in the specification is observed and the machine can move to state *S1*. Similarly, the state machine stays in the state *S1* until the method *mergeSort.sort()* starts executing. When this method starts executing the state-machine reaches the accept state; that is, the specification is supported by the UML models.

Although the control language supports wildcard transitions, our implementation does not make use of them. The reason for that is that the execution semantics of most of the sequence diagram actions are modeled using more than one graph transformation rule that are applied in an order. Without a control automaton, such an ordering is achieved by giving priorities to the rules. With a control automaton, prioritization is not possible, so the control automaton should also include the rule ordering (with the VSL specification). Figure 3.19 shows the state-machine of Figure 3.18-(b) with the wildcard replaced by the actual actions that are simulated. Here, the transitions whose labels end with *()* are state-machines that order the transformation rules modeling the semantics of a sequence diagram action; we call these transitions *action functions*. For example, the transition *returnframe()* is a state-machine that simulates the return action. Due to space limitations, the figure only presents the *createObject()* action function, the *returnframe()* action function and the *methodCall()* action function (some call action functions and the reconfiguration action functions are omitted).

From the action functions, the transition *methodCall()* is detailed in Figure 3.20. The *methodCall()* action function provides an order between the graph transformations presented in Figure 2.14 and Figure 2.15.

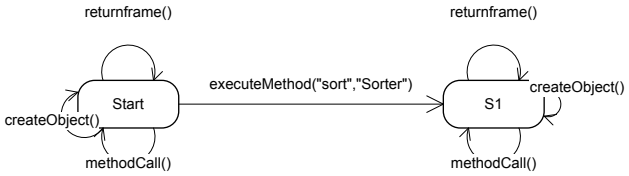


Figure 3.19: The state machine showing the action functions of each state

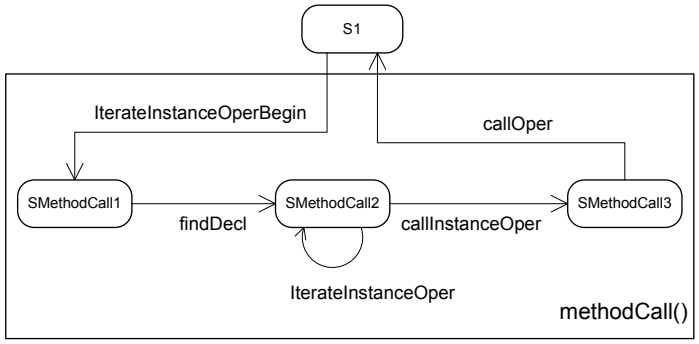


Figure 3.20: The state machine detailing the action function *methodCall()*.

In summary, the action functions are repeatedly applied at a state S_i conforming to a State-VSL transition VT_i. Assume that the the transition VT_i connects to the state $VS_{(i+1)}$; then, the continuous application ends when the simulation action specified at the State-VSL state $VS_{(i+1)}$ is observed. In the state machine form of the State-VSL specification, this state conforms to the transition $T_{(i+1)}$. One way to express this behavior in GROOVE's control language, is using functions to represent the states and the transitions of a state machine form of a State-VSL specification. The repeating application of the action functions can be achieved using the infinite loop *while(true)*. The action functions are implemented as control functions; this allows the grouping/ordering of the transformation rules of an action function. The control functions representing the transitions of a state-machine form of a State-VSL specification consist of two statements: the first statement is the desired simulation action (e.g. *executeMethod*) and the second statement is the call to the function representing the next state. Thus, in the while loop, the control program should instruct the transformation engine to try to apply the action functions and, if they fail, to try to apply the functions of the transitions. Below, the template of the control functions of a state and a transition implementing the described behavior is presented:

1: function $State_i$(){
2: while(true) do{
3: try{
4: methodcall() | thiscall() | createObject() | returnAction() | superCall() | poly-

```
    morphicReconfiguration() | conditionalReconfiguration() | dynamicTypeLoad-
    ing();
 5: }
 6: try{
 7: SimulationAction;
 8: Transition_i() | Transition_{i,2}() | ...;
 9: } } }
10: function Transition_i(){
11: SimulationAction;
12: State_{(i+1)}();
13: }
```

Here, the lines $4-5$ are the calls to the action functions. These calls are in a *try* block; thus, if they cannot be applied the control program jumps to line 7. At line 8, the calls to the control functions representing the outgoing transitions are implemented. In case there is a single outgoing transition then there is only one function call without the or operator |. These calls are also within a *try* block meaning if the transition cannot be applied then the control program returns to line 4. The lines $11-12$ shows the statements of a transition function. Here, *SimulatioAction* represents the simulation action to be observed; we call these action the *observation actions*. The conversion from State-VSL to control program generates an instance of the control function $State_i$ for each transition and an instance of the control function $Transition_i$ for each state in the specification (except the states representing the start and end of the program). An instance of a function $State_i$ is created by specifying the outgoing transitions at line 8 and an instance of a function $Transition_i$ is created by specifying the simulation action to be observed at line 12 and specifying the state this observation yields to at line 13. Note that these control programs only instruct the transformation engine to apply a series of rules at a certain state. If none of these rules can be applied, the simulation stops. Thus, the infinite loops used by the functions shown above are only used to instruct the simulator to apply a series of rules continuously. If, for example, there is a null pointer in the DCM or there are no more call actions to simulate, the simulation terminates. Algorithm 1 presents the control program generated from the State-VSL specification shown in Figure 3.18-(a).

The following observation actions can be specified in a State-VSL specification:

- *executeMethod_< methodName >_< object-type name >*: to observe the beginning of an execution of a method.
- *returnFrame_< methodName >_< object-type name >*: to observe the return from a method.

Algorithm 1 The control program for verifying the State-VSL specification presented in Figure 3.18-(a)

```
1: startexecution;
2: State0();
3: function State0(){
4: while(true) do{
5: try{
6: methodcall() | thiscall() | createObject() | returnAction() | superCall() | polymorphicReconfiguration() | conditionalReconfiguration() | dynamicTypeLoading();
7: }
8: try{
9: Transition0();
10: } } }
11: function Transition0(){
12: executeMethod_Sorter_sort;
13: State1();
14: }
15: function State1()
16: while(true) do{
17: try{
18: methodcall() | thiscall() | createObject() | returnAction() | superCall() | polymorphicReconfiguration() | conditionalReconfiguration() | dynamicTypeLoading();
19: }
20: try{
21: Transition1();
22: } } }
23: function Transition1(){
24: executeMethod_quickSort_sort;
25: accept
26: StateFinish()
27: }
28: function StateFinish(){
29: while(true) do{
30: try{
31: methodcall() | thiscall() | createObject() | returnAction() | superCall() | polymorphicReconfiguration() | conditionalReconfiguration() | dynamicTypeLoading();
32: }
33: try{
34: finish;
35: } } }
```

- *polymorphicReconfiguration_< to object-type name >_< from object-type >*: to observe polymorphic reconfiguration from an instance of an object-type to an instance of another object-type.

- *conditionalReconfiguration_< conditional variable >_< value >*: to observe the conditional reconfiguration when *conditional variable == value*. Here, *value* is a user specified string and, thus, it can refer to a range of values like < 4.

All these actions are transformation rules that do not modify the DCM but only match when the simulation is at the stage they are observing. A major drawback of the control program is the lack of parameter support. In GROOVE, one can use the feature *transition parameters* to extract information from the graph about where a rule has matched. The transformation rule *executeMethod* described in Section 3.3 uses this feature to output the name of the method the simulation started executing. However, the feature transition parameters work only in one-way: one cannot specify the values for the parameters before the simulation and restrict the rule to match only to the nodes with those values. This behavior is required for the semantics of the transition to the next state to work in the state-machine form of a State-VSL specification. For example, assume we want to verify the execution of the method *foo.mbar()* using State-VSL, then, we need to be sure that the simulation has executed this method. The state-machine ensures this with transition *executeMethod_< mbar >_< foo >*, but in order for this transition to be triggered, we need a rule which only matches when an operation frame is pointing to an object-type node named *foo* and to an operation implementation node with a signature named *foo* during simulation. To ensure that the observation actions match at the right locations during simulation, we modeled the transformation rules of observation action as templates. In these template rule the names of DCM elements, like the name of an object-type, are parameterized. Figure 3.21 presents the template of the rule *executeMethod*. Here, the names *@typeName* and *@methodName* are the parameters. During the generation of the control program, these strings are automatically replaced by the user supplied names. For example, for the VSL state *executeMethod("sort","Sorter")*, the converter first creates a new transformation rule named *executeMethod_Sorter_sort* and, then, replaces the name *@typeName* by the string *Sorter* and the name *@methodName* by the string *sort*.

The Accept and the Reject States

An important aspect of the state machines used for expressing reconfiguration requirements is the notion of accept and reject states. The generated state-space has no notion of accept and reject states; it only has a start state and final states. The

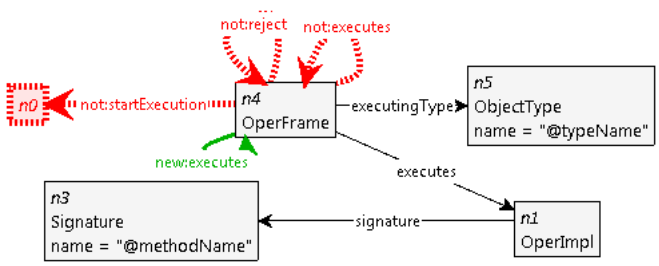


Figure 3.21: The template for the transformation rule *executeMethod*.

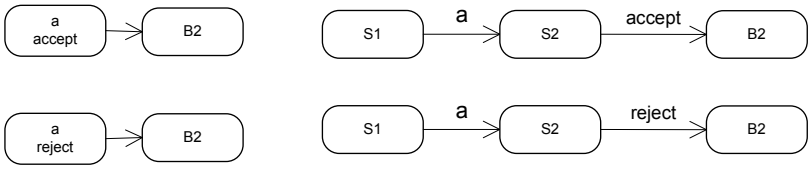


Figure 3.22: The conversion of State-VSL accept and reject states to state-machine transitions.

support for accept and reject states can be added by making them graph transformation rules. The application of these rules add transitions labeled *accept* or *reject* to the generated state-space. The transformation rule *accept*, for example, adds an edge to the current operation frame. Because a DCM always contains an operation frame, this rule always matches in the free-form application. When the control automaton is generated from the VSL specification, the matching of this rule is restricted such that it matches only when the simulation reaches an accept state.

An accept or a reject state VS_i in State-VSL specification is converted to a transition T_i which moves the system to an intermediate state S_ia. The transition T_i has the same simulation action as the State-VSL state VS_i. The intermediate state S_ia has one outgoing transition (i.e. no self transitions with wildcards) with labels *accept* or *reject*. Thus, when the simulation action specified in the State-VSL state VS_i is observed, the system moves to the state S_ia. Because at this state there only one transition to the next state with labeled *accept* or *reject*, the simulation is restricted to apply the transformation rule conforming to the labels; for example, the transformation rule *accept* is applied if the transition is labeled *accept*.

In the VSL specification of Figure 3.18-(a), the state *executeMethod("sort", "mergeSort")* is an accept state. Thus, the transformation rule *accept* should only match (once) after the transformation rule *executeMethod("sort", "mergeSort")*. Figure 3.18 shows how this is expressed in the control automaton, where the accept state is divided into two states *S2* and *accept*. The only transition between these two states is the transition *accept*. As a result, after applying the transformation rule *exe-*

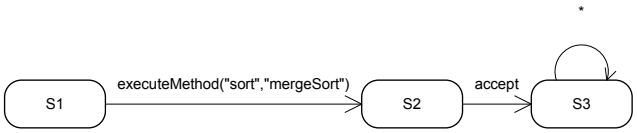


Figure 3.23: The state machine that adds support for the accept state

cuteMethod, the transformation is only allowed to apply the transformation rule *accept*.

Because accept and reject are properties of State-VSL states, they are represented in the transition functions of the control program conforming to the State-VSL specification. That is, a control function representing the intermediate state from which the accept or reject transition occurs is not placed into a separate function but placed within the transition that represents the State-VSL state. A transition function representing an accept/reject state, consists of three statements: the observation action that triggers the transition, accept or reject, then the call to the control function representing the next State-VSL state. The State-VSL specification presented in Figure 3.18-(a) goes to an accept state when the execution of method *QuickSort.sort()* is observed. The transition function conforming to this state is the control function between lines $23 - 27$ in Algorithm 1. Here, the statement *executeMethod_sort_quicksort* is the observation action which triggers the transition from the state *State1* to *StateFinish*. After this action, the control program states that the transformation engine is only able to apply the rule *accept*. This is the only place the transformation engine is allowed to apply the transformation rule *accept*, and applying this rule adds the transition labeled *accept* to the generated state-space.

Verifying the Existence of Reconfiguration

For the first type of reconfiguration requirements, if the simulation yields an execution sequence with an accept state after which there is no reject state then the requirement is supported by the UML models. The error diagnosis mechanism uses the following CTL formula to verify the requirement:

$$EF(accept \wedge !(EF(reject)))$$

If there are states that satisfy this formula then the reconfiguration requirement is supported. On the other hand, if there are no such states then the reconfiguration requirement is not supported. In this case, similar to the error diagnosis mechanism based on CTL, the error diagnosis mechanism outputs the point up to which the VSL specification is supported. However, this error diagnosis mechanism outputs

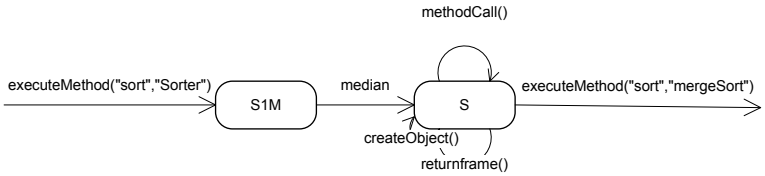


Figure 3.24: The state $S1$ in Figure 3.18-(b) extended with the median transition

the whole execution trace (i.e. includes all the methods/reconfiguration mechanisms that executed between VSL states).

The error diagnosis mechanism uses *median* transitions to provide such an output. When the execution simulation reaches an intermediate state, the transformation rule named *median* is allowed to match (similar to accept and reject transformations, this rule also always matches to the DCM in free-form application) and the application of this rule adds a transition labeled *median* to the state-space. With the CTL formula $EF(median)$ the execution sequences that yield at least one of the intermediate states are found. The error diagnosis mechanism outputs the important transitions (e.g. method calls and reconfigurations) until it reaches a state that does not satisfy this CTL formula (i.e. a counterexample) in each execution sequence (or branch of the state-space).

Figure 3.24 presents the state machine that adds the median transition to the intermediate state $S1$ of Figure 3.18-(b). Here, after the transformation rule *executeMethod*, only the transformation rule *median* is allowed to match. The VSL to control automata converter adds the median transition to all intermediate states.

Verifying the Reconfiguration Invariants

For invariants, the VSL specification expresses an execution sequence that violates the invariant. If this execution sequence is supported by the UML models, then the models violate the invariant. So, if the simulation yields an execution sequence with an accept transition after which there is no reject transition, then the invariant is violated. The CTL formula $EF(accept \wedge !(EF(reject)))$ is used for finding the execution sequences that violate the invariant.

If states that satisfy the formula presented above are found then the invariant is violated and the trace of the execution sequence that has this violation should be presented to aid the designers. The states from which an accept transition (after which there is no reject transition) is reachable from the execution sequence that violates the invariant. The error diagnosis mechanism, for each execution sequence, outputs the transitions that show method/reconfiguration executions until it reaches

a state that belongs to the counterexamples.

The Limitations of Error Diagnosis based on Control Automata

Some reconfiguration requirements require reasoning about the execution of two methods after a method finishes execution. For example, a reconfiguration requirement may state that after a eventually both b and c execute. Such requirements can be expressed in CTL and VSL (using transition forks). However, they cannot be expressed with state machines without providing an order between b and c.

The state machines also do not support the VSL *all* transitions. These transitions translate to the CTL operator AF which allows one to reason over all branches.

Another limitation of using control automata is that one specification can be used for one state-space. With CTL, the whole state-space is generated and verified. With control automata, the simulation generates the state-space according to that specification. Thus, to verify another specification the simulation has to be repeated for that specification.

3.7 Related Work

This section discusses the research that is related to the contributions of this paper. For ease of understanding, we categorized these works according to the contribution that they are related to.

3.7.1 Reconfiguration Requirement Verification through Graph-Based Model Checking

Our approach specializes graph-based model checking for the verification of reconfiguration requirements. In the literature, obviously, different approaches and processes have been developed for the verification of requirements with respect to different software artifacts. These approaches can be grouped into 3 groups according to the software artifact they operate with. Below, the work that is related to reconfiguration requirement verification in these 3 groups is listed and compared with our approach.

Software Architecture Models: It is important to correctly specify the reconfiguration, otherwise the software system can exhibit undesired combinations of its

functionality. For example, a wrong specification in a configuration script may cause the software to crash. The actual implementation of the software may be too detailed to see which parts of the components will be used for a configuration, which in turn may cause one to specify the reconfiguration wrongly. An abstraction showing only the parts of the software related to the reconfiguration is more desirable in reconfiguration specification as the unrelated details are not shown to the stakeholder. In the literature, three types of models are proposed for reconfiguration specification. The first type uses component models where the reconfiguration specification changes the communication between components [107, 88]. The second type uses product line models to specify a reconfiguration of the system [130, 66]. The third type uses aspect oriented modeling to model the variability points of the application; the reconfiguration specification selects an option from all the possible options depending on the context of the application [103].

Without formal specification of the reconfiguration, the specification may be incomplete (i.e. it might not include all the conditions of the reconfiguration). To address this problem, formal specifications for reconfiguration are proposed [24]. For example, Wermelinger et al. [125] use a language that is an abstraction over the *CommUnity* program design language. In this language, a program consists of a type, its values, boolean expressions on these values and statements that are executed when the boolean expressions are true. By defining morphisms on this language it is possible to superimpose a program. Aguirre et al. [13] propose a specification language for architectures that uses temporal logic to formally specify the adaptations of the system.

All studies listed above aim at precise/correct specification of the desired reconfigurations. They do not provide support for verification whether an application that is to be reconfigured can support the specified reconfiguration. An application may fail to reach the desired reconfiguration even though the specification is correct because the application may crash due to errors. We take the interactions between application objects specified as sequence diagrams as input and simulate these interactions with the possible reconfiguration actions. This simulation can capture errors that are caused by reconfiguring the interactions between the objects. Such errors cannot be captured by the approaches listed above. In our approach the reconfiguration actions are triggered by the configuration system, which is responsible for sending the correct values to the application. Our approach does not take into consideration the configuration system; we only focus on the application and how the configuration system interacts with the application. Thus, our approach can be used to complement the approaches listed above such that application errors are also covered in reconfiguration specification.

Usually, scenario-based analysis methods are used to evaluate the software architec-

tures with respect to quality attributes [80, 21, 79, 91]. In all scenario-based analysis methods, the evaluation is done by the architects, which can lead to subjective and incomplete evaluations. In our approach the simulation generates all possible execution paths supported by the design and the evaluation is done automatically. This eliminates the subjectivity and incompleteness of the manual evaluation approaches.

Bucchiarone et al. [25] provide graph grammars for modeling the reconfiguration architectural models and show how these rules can be implemented in Dynalloy [57]. Similar to our approach, these rules can be simulated on the models of the application and properties of the reconfiguration can be verified. The problem with these rules is that they are application-specific. To apply them to a different application, one can use these rules as guidelines and implement rules specific to the new application. This an extra required task after modeling the application. Besides the application-specific rules, the level of details in the architectural model proposed by the authors is not sufficient to model OO designs. The inputs to our approach are UML models which are developed for modeling OO designs and capturing the interactions between objects. The transformation rules used for simulating the sequence diagrams are not application-specific and they can be used to simulate any UML models provided that they are modeled correctly.

UML models: The UML is a widely used standard modeling language for OO software design. Because of its popularity and extensibility, variations of UML models are proposed in the literature for modeling special types of systems, such as mechatronic [69] and real-time systems [16]. These models are extended with formal verification tools that specialize on verifying runtime reconfiguration of these systems. We identified three drawbacks of these verification techniques; below these drawbacks and how our approach addresses these drawbacks are detailed:

- *Additional models are required besides the UML models for verification.* Giese et al. [60] provide a reconfiguration verification that can check the inconsistencies arising from parallel executing components on mechatronic systems. The structural input for this model are the UML component models and the behavior of the system is modeled with real-time state-charts. In addition to these models, one also needs to provide the following models for the system: *hybrid reconfiguration automaton*, showing the inputs, outputs and I/O conditions on the transitions, and *interface state charts* showing the external behavior of the component. Although such I/O models are required to model the hardware/software interactions, they are an extra burden for the designer when the focus is shifted to the software side. The conditions for reconfiguration are already present in the sequence diagram: for example, for frame fragments one needs to specify the guards, or for polymorphic reconfiguration these conditions are all the compatible types that can receive the call. Thus, our approach

requires no additional behavioral model of the system to be specified expect the sequence diagrams. In addition to this, we use the UML meta-model as is and extend it with only two annotations for behaviorial modeling.

- *Execution/Reconfiguration semantics should be provided with UML models.* To verify that any structural configuration of mechatronic systems follows the safety requirements of the system, Becker et al. [19] propose simulation of the system using graph transformations. Here, one needs to model application-specific execution semantics with graph transformation rules and the unsafe states the safety requirement addresses are modeled through application-specific graph patterns. Application-specific semantics provides an evaluation that is at a high level of abstraction where the detailed design of the software is not yet present. However, at the OO design phase, the structure of the software is realized and the interactions between objects are modeled. Modeling application-specific execution semantics at this phase is an extra task because the runtime semantics are generic to object-oriented software. Thus, with our approach, the user is not required to provide the execution semantics. We provide generic execution/reconfiguration semantics that can be used to model any UML sequence diagram provided that the UML models do not contain errors.

  The TURTLE UML profile provides extensions to UML for modeling real-time systems [16]. The formal semantics of the TURTLE UML is expressed in the RT-LOTOS and any TURTLE UML model can be converted RT-LOTOS for formal verification. Apvrille et al. [15] shows how TURTLE UML can be used for modeling the dynamic reconfiguration of the software systems and verifying the continuity of the system during reconfiguration. In this application, the authors provide execution semantics of communication between components and reconfiguration. These semantics should be copied and specialized when one needs to use this approach. Thus, in this approach one still needs to provide execution/reconfiguration semantics with the models of the software system. In our approach, the execution/reconfiguration semantics are hidden from the user. The user is not bothered with providing execution/reconfiguration semantics and, thus, the user only needs to specify the interactions between objects using sequence diagrams.

- *Execution semantics not suitable for verification on detailed OO designs models.* The approaches provided above use UML meta-model elements to represent component-based architectures where the behavior of the component is modeled using an extended state-chart and the verification system checks whether the right transitions are triggered when the conditions for reconfiguration evaluate to true. The component architectures are a different level of

abstraction than detailed OO design and, as a result, these semantics become inadequate for verification of the reconfiguration requirement on detailed OO design models such as UML models. At the OO design phase, the interactions between objects play a crucial role on how the software reconfigures as the reconfiguration mechanisms changes these relations. In addition to this, the verification on these models should be realized with execution semantics that are very close to the actual execution of the OO software. The graph transformations presented in this paper are designed to simulate the effects of the reconfiguration on the call relations between objects which is the type of the interaction that can be modeled in sequence diagrams. While modeling these transformations rules, we are inspired from execution specifications of different OO languages [9, 27]. Thus, the execution semantics we provide are suitable for verifying reconfiguration requirements on the UML models of OO-software.

OO-Programs: The programming languages have well-defined syntax but their execution semantics are informally specified. In the literature, there are approaches that define formal execution semantics for OO-programs to overcome this problem to provide model checking at the implementation phase [123, 76]. Visser et al. [123] propose the *Java Path Finder* (JPF) model checker for Java programs. In this approach, the model checker is a custom virtual machine that provides user-guided Java bytecode execution. Such a model checker is not suitable for our purposes, because the runtime state-space extractor depends on the values provided by the user. This causes the model checker to generate state spaces for *popular* execution traces. The problem, here, is that these popular execution traces may not cover all possible reconfigurations of the software system and the designers may not actually know about these traces. For example, the designer knows the values required to generate execution traces that are explicitly modeled with sequence diagrams. However, the problem is with the traces that are formed by a combination of the sequence diagrams which are implicit and they can better be exploited with full state-space generation. Kastenberg et al. [76] model execution semantics of the TAAL language (a simplified version of Java) as graph transformations. Here, the idea is that a program in the TAAL language can be compiled into a graph model and simulated using graph transformation rules. By using graph-based model-checking, the properties of the execution can be verified.

Both of these approaches provide full-semantic simulation of the OO-programs, which means that every statement in the program is simulated. This may in turn generate too large states-spaces compared to only simulating the call and return actions as we do in our approach. Besides this, a much more important problem regarding the use of model checking OO-programs is that they require a complete executable program which hampers the usability of the approach for early evaluation.

Even though a UML-to-OO-program converter can be implemented, one still needs to provide all the values of the attributes, parameters and/or variables used in the input sequence diagrams. This an extra task required after modeling the design of the software and, moreover, at this phase the actual values of all the attributes may not be known. Whereas, in our approach, the designers are only required to give a name representing the values. The case study in section 4.1 shows that this level of information is adequate to find some problems in the interactions between the objects of the application related to reconfiguration. The simulation also shows which values the configuration system should provide to prevent an invariant violation or to allow a reconfiguration. However, our approach does not help in checking these values once the design is implemented. This is because we tailored the approach to provide model-checking with available information in the OO design phase.

Due to the complexity of the OO programs and the lack of models for representing references between objects, reference errors can be introduced to programs. Distefano et al. [43] address this problem by providing a reference model where abstract reference models can be generated by combining similar items. Because polymorphism deals with changing the references between objects, it may be argued that this reference model can be used instead of the generated state-space. This model is, however, static and lacks the call relations that lead to the change in the references. For example, a reconfigured object r may refer to another object o; however, if r does not call a method of o then it may violate an invariant. Thus, the call relations with the references are required for verification of the reconfiguration requirements. The state-space generated by the simulation, on the other hand, shows the calls between objects and because these calls are simulated they are based on the references defined in the sequence diagrams. Besides the lack of call relations, another limitation of this model is that only linked-list structures can be represented. Such a limitation does not exist with the generated state-space and the calls to all references of an object can be simulated.

Similar to a sequence diagram, a call graph of an OO program shows the possible receivers of a call from a method [63]. These possible receivers are computed by looking at the types of the variables, just like the semantics we provide for polymorphic reconfiguration and for method dispatch. Due to these similarity, it may be argued that it would be sufficient to combine the sequence diagrams to form the call graph. Our approach actually realizes this combination; however, it takes the call graph construction a step further and exercises the actual dispatches of the calls with the simulation. The simulation is a required step, as reconfiguration changes the execution order of objects. For example, an object k may depend on another object d created by the object l. Thus there is an implicit execution order between the object l and k; first the object l should execute than the object k should execute. Now assume that a reconfiguration happens and this order changed, where

the object k is now executed before the object l. This would cause a null pointer exception in the execution of the object k, which can only be captured by simulating the call actions.

3.7.2 Visual State-Base Language for Expressing Reconfiguration Requirements

CTL, or, in general, Branching time Temporal Logic, is a widely accepted specification language for execution sequences. During our case study, we noticed that designers are reluctant to use CTL to express execution sequences. This is mostly due to the effort necessary to understand the underlying formalism. This is in fact a widely studied problem in the literature as such reluctance hampers the adoption of CTL to practical cases. For example, Bimbo et al. [41] provide a visual language to make it easier to specify Branching time Temporal Logic. The major drawback of this language is that, one still needs to know that propositions are composed with logic operators and to specify, for example, after all the underlying tree structure. Besides this, the language does not use familiar notations; so, one needs to develop tools to use this language.

We addressed the problem of the usage of CTL with a visual language called *VSL*. The VSL is tailored for expressing an execution sequence and does not cover all the semantics of the CTL. Because of this, VSL hides the underlying tree-structure and the logic operations. For example, one never needs to specify logic operators like *and* in VSL. We chose to use state-based semantics for VSL because the familiarity of the designers/developers to such specifications and the availability of state-machine editors in UML tools.

The temporal logic based specifications of the reconfiguration of the system lack the description of adaption semantics. For example, they describe the states of the program before and after reconfiguration; however, they lack in specifying when this transition is going to happen. Zhang et al. [131] address this problem by extending Linear Temporal Logic. In this extension, one also specifies the type of transition while specifying the transition of the states. Here, the types describe the semantics of when the state transition occurs. In our approach, the *when* aspect of the reconfiguration is implicitly covered by the choice of the reconfiguration mechanism. For example, with polymorphic reconfiguration one achieves *overlapped adaptation* where the reconfiguration can change the receiver of a call while another receiver is executing. Because of this implicity, one can specify the reconfiguration mechanisms when expressing a reconfiguration requirement as an execution sequence in VSL or CTL.

3.7.3 Runtime reconfiguration mechanisms

Most techniques on runtime reconfiguration of object-oriented systems focus on the use of reflection and design patterns [81, 28, 45]. Obviously, our aim in this chapter is not to propose a new runtime reconfiguration mechanism but to verify whether a desired runtime reconfiguration can be reached by the software using these mechanisms. We chose to model polymorphism, conditional statements and dynamic type loading as reconfiguration mechanisms because these mechanisms are supported by many languages. Nonetheless, the graph-based approach is expressive enough to model configurations that can be created by the help of design patterns and language-specific reflective techniques. These can be introduced to our approach by modeling their semantics as transformation rules.

3.8 Conclusions and Future Work

This chapter provides a process for formally verifying that the interactions between objects specified with UML models are correct with respect to runtime reconfiguration requirements. In this process, the UML models are converted into a graph-based model. With graph transformation rules that model the execution and reconfiguration semantics of UML models, the execution of the models is simulated. The simulation results in a transition system showing the applied reconfiguration mechanisms and the executed objects/methods. Runtime reconfiguration of OO software causes changes in the execution sequence of objects/methods and this changed execution can be used to verify the requirement. The requirement engineer specifies reconfiguration requirements and expresses the execution sequences conforming to these requirements as CTL formulas. The verification algorithm searches the state-space resulting from the simulation to find states from which the execution sequence complies with the sequence given in a CTL formula.

Polymorphism, conditional statements and dynamic type loading are the reconfiguration mechanisms used in the software developed by our industrial partner and we modeled these mechanisms by graph transformation rules. If additional reconfiguration techniques are needed to be modeled, they can be added by using the graph editor of GROOVE. In fact, we are currently working on graph transformation rules that simulate reflective techniques. An important aspect of these graph transformation rules is that they match to the DCM when the UML models contains certain tags. In this way, the UML models do not need to specify the implementation specific details of how the reconfiguration is achieved. It specifies the reconfiguration mechanism that will be implemented and checks whether the entities required

by the mechanism are present in the UML models. To improve the scalability of the approach, the transformation rules modeling the reconfiguration semantics are designed to match to the DCM when the execution reaches a statement where a reconfiguration can be applied. These rules can also applied before starting the execution simulation; that is, first branches showing each reconfiguration the design supports are generated, and then each branch is simulated. However, this generates branches that show invalid reconfigurations in GTS.

We are currently developing a UML profile for reconfiguration such that the reconfiguration mechanisms like polymorphism and dynamic type loading are supported by the UML meta-model. We also plan to develop plug-ins for popular UML editors to integrate the UML to DCML converter.

Chapter 4

Evaluation of the Reconfiguration Requirement Verification Process

The previous chapter introduced a process for verification of reconfiguration requirements on UML models. In this process, the UML class and sequence diagrams are converted to a graph based-model and the execution of these models is simulated using graph transformation rules modeling the reconfiguration and execution semantics of object-oriented software. This simulation generates a state-space with transitions showing all the methods and reconfiguration mechanisms that are executed; the branching in this state-space is caused by reconfiguration. The execution sequence that conforms to the reconfiguration requirement is expressed as a CTL formula and a verification algorithm searches the generated state-space to find whether there are states that satisfy the formula.

In the next section, a case study conducted with a designer from the industry is provided. In this case study, the reconfiguration requirements of the current version and the expected near future reconfiguration requirements are verified on the UML models of an industrial software called Data Monitoring Viewer (DMV). The result of the verification is compared with the actual implemented software: the aim of this comparison is to show that the simulation and the execution of the actual software agrees.

The case study with the designer from the industry showed that CTL is hard to be used by the designers and a detailed feedback is required when the reconfiguration requirement is found to be not supported by the models (i.e. a feedback on the possible location of the problem is desired). To address the first problem, we designed a visual state-based language called VSL (Section 3.5). To address the second problem, we designed two feedback mechanisms: one based on CTL (Section 3.6.1)

and the other one based on control automata (Section 3.6.2). To test how effective these mechanisms are, we conducted two experiments with computer science master students from University of Twente. Section 4.2 details the experiment on the effectiveness of the CTL based feedback mechanism and Section 4.3 details the experiment on the effectiveness of the control automaton based feedback mechanism. In both experiments, the UML models of the DMV tool is used.

4.1 Case Study with Designer from the Industry

We have applied the process for verification of reconfiguration requirements on UML models for verifying the reconfiguration requirements (current and expected near future) of an industrial software called Data Monitoring Viewer (DMV). This software is used for displaying the received signals of an MRI machine and it allows the user to manipulate certain parameters to display the effects on the signal. The objective of this case study is to compare the verification results on the UML models with the implemented software. This case study is conducted together with the designer of the DMV.

DMV is written in C# (21K LOC, 139 classes) and is designed to be functionally extendable; it can be extended with signal viewers for specific signals, with user interface elements (e.g. buttons) that add extra functionality (e.g. loading the waveform from a file). Although DMV is a stand alone software, there are also extensions that integrate the software to the MRI software framework so that it can interact with the MRI machine. These extensions are implemented in a class and these classes are compiled as Dynamically Linked Library (*dll*) files. These dll files are placed in a specific directory; when DMV is launched, before displaying the main window, it scans this directory and loads the classes where the extensions are implemented from the dll files. The class structure of DMV is similar to the decorator pattern. The classes are divided into two groups: interface providers and program decorators. The interface providers implement the interfaces (there are 6 interfaces) that can be used by the program decorators to add new functionality; every extension is a program decorator. In this section, we first describe the design of DMV, then we present *8* reconfiguration requirements that are verified with the process, and, lastly, we list our conclusions.

4.1.1 Design of the Data Monitoring Tool

Figure 4.1 presents a sequence diagram of the DMV tool, represented by the class *Main*, with one interface provider, the class *PE* that implements the interface *No-*

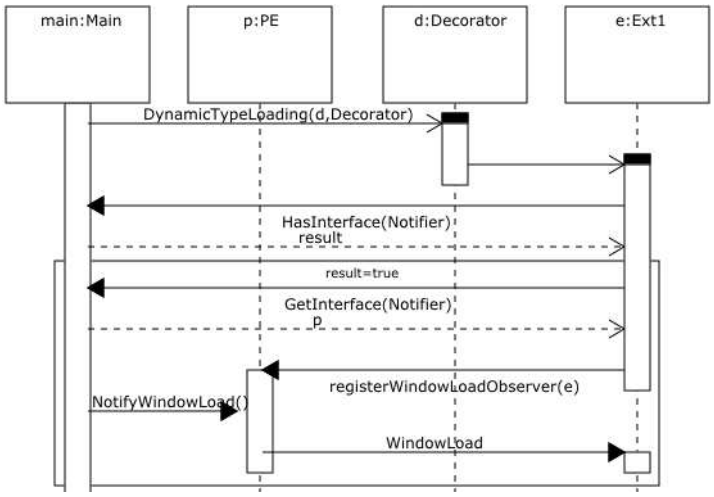


Figure 4.1: Sequence diagram showing an extension attaching itself to the signal viewer.

tifier, loading the extension implemented in class *Ext1*. This sequence diagram presents a stripped execution sequence and the methods provided in the sequence diagram do not reflect the methods in the actual UML models (e.g. the parameters of the methods are unrealistic). We present this sequence diagram to give an overall impression of how the DMV works. The sequence diagram starts with the class *Main* loading the class *Decorator* and calling the constructor of this class; the constructor of this class in turn creates an instance of the class *Ext1*. The constructor of the class *Ext1*, by calling the method *HasInterfaces*, asks if the instance of *Main* encapsulates an object which implements the interface *Notifier*. The interface *Notifier* declares methods that are used for the notification of extensions when the execution reaches certain points, e.g. the point where the main window becomes visible. The class *PE* implements this interface. In this sequence diagram, the instance of the class *Main* encapsulates an instance of the class *PE*; so, *Main* returns true. The extension then gets the instance of *PE* from *Main* and attaches itself as a window load observer (by calling the method *PE.registerWindowLoadObserver*). When the class *Main* notifies about the window load by calling the method *PE.NotifyWindowLoad*, the method *Ext1.WindowLoad* is called.

For the case study, we used the class diagram of the DMV tool and 4 sequence diagrams showing the successful execution of 4 extensions (Ext_1, Ext_2, Ext_3 and Ext_4). The sequence diagrams in total contain 66 call/return actions. The sequence diagram in Figure 4.2 presents all the calls related to the loading of the extension Ext_1; every sequence diagram used in this case study start with these calls (note that because lack of conditional statement support in ArgoUML, the dynamic type loading is shown as the self-call *DynamicTypeLoading()*). In addition to loading of

the extensions, the sequence diagrams show how each extension attaches itself as a window load observer. Each extension uses two reconfiguration mechanisms: dynamic type loading (used when an extension is loaded from a dll file) and conditional reconfiguration (used by the extensions to test the interfaces the loader program can supply). From these UML models, a DCM consisting of 533 graph elements is generated using the UML-to-DCML converter. The state-space generated after the simulation is a tree with more than 120 branches and consists of 22579 states and 22578 transitions. The simulation took 1.6 minutes and 23MB memory, including the time and memory required to draw the state-space to the UI, with Intel Centrino 1.7GHz processor, 1GB memory and running Windows XP. The simulation generated this many states because DMV supports many reconfigurations. Each type loading statement has 5 possible reconfigurations (there are 4 of them in the sequence diagram): loading one of the 4 extensions (it is normal to load the same type more then once) or failing. Moreover, for each extension there are at least 2 reconfigurations: the class *Main* can/cannot supply the interface providers needed by the extension.

4.1.2 Verified reconfiguration requirements

The requirements engineers provided us with the list of runtime reconfiguration requirements the current version of the DMV tool should support. Besides these requirements, they also provided two reconfiguration requirements that they plan to deploy in the coming version of the software. We verified these requirements using graph-based model checking and by manually testing with the DMV to see whether the simulation of the UML models agree with the actual implemented version of DMV. Table 4.1 summarizes the verified requirements and the outcome of the both verification mechanisms. Below the verification procedure is detailed (for some requirements we present the CTL formulas with only two extensions due to complexity of the formulas):

- *DMV should support an execution where none of the extensions executes; the method NotifyWindowLoad is run but none of the extensions execute.*

  This requirement is verified with following CTL formula:

$$EF(executeMethod(\text{"}NotifyWindowLoad\text{"}.\text{"}PE\text{"})$$
$$\wedge !(EF(executes(\text{"}ext1\text{"})))\wedge !(EF(executes(\text{"}ext2\text{"}))))$$

  This formula looks for states where *PE.NotifyWindowLoad* starts execution and afterwards neither of the extensions has received a call. The verification of this

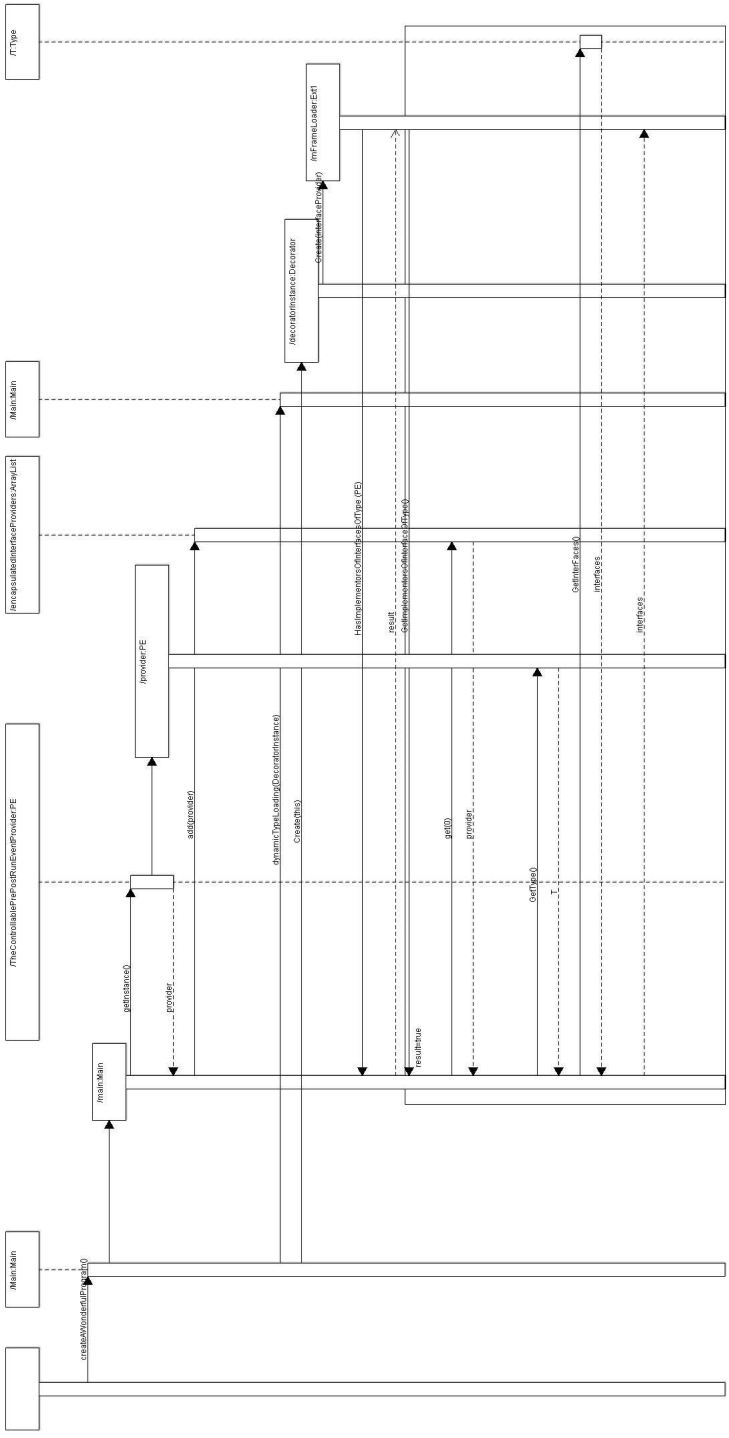


Figure 4.2: The sequence diagram showing all the calls involved in loading the extensions *ext*1 to the signal viewer tool.

Table 4.1: The requirements of verified using graph-based model checking on the UML models of DMV and manually executing the DMV tool. The outcome of the both verification techniques agree.

Requirement	Description	Results of Verification on UML diagram	Results of Verification by manual execution
R_1	DMV should support execution with no extensions	Supported	Supported
R_2	DMV should not stop when loading of an extension fails	Supported	Supported
R_3	Extensions should not register as window load observers when interfaces they require cannot be supplied	Supported	Supported
R_4	Extensions should register as window load observers when interfaces they require can be supplied	Supported	Supported
R_5	Extension Ext_3 should not register as a window load observer if process framework is not available	Supported	Supported
R_6	DMV should support extensions that depend on other extensions	Not Supported	Not Supported
R_7	DMV should support more than one provider for the same interface	Supported	Supported
R_8	All extensions should execute with the same provider	Not Supported	Not Supported

formula succeeds; this means that the UML models supports this requirement. We executed the tool without extensions and observed that the DMV tool was also able to run without extensions.

- *DMV should not stop executing when dynamic loading of one of the extensions fails.*

  If the simulation yields an execution sequence where dynamic type loading fails (for any extension) and the method *NotifyWindowLoad* is not executed, then the requirement is not supported. The following CTL formula looks for this execution sequence:

$$EF(DynamicTypeLoading\_fail() \wedge \\ !(EF(executeMethod("NotifyWindowLoad"."PE"))))$$

  The verification did not find any states that satisfy this formula, meaning that the UML models of the DMV tool support this requirement. When we executed the DMV tool with *dll* files that do not contain the types the tool tries to load, the tool did not crash; the implementation and simulation of the UML models agrees for this requirement.

- *For all extensions, the method WindowLoad should not execute when the interfaces it requires cannot be supplied.*

  The execution sequence of this requirement for an extension is formulated in CTL as follows:

$$\begin{aligned} &EF(executeMethod("ext_i"."ext_i") \wedge \\ &(EF(conditionalexecutes("false"."result") \wedge \\ &(EF(returnframe("ext_i"."ext_i") \wedge \\ &!(EF(executeMethod("ext_i"."ext_i"))) \wedge \\ &(EF(executeMethod("NotifyWindowLoad"."PE") \wedge \\ &(EF(executes("ext_i")))))))))) \end{aligned}$$

  This formula searches for states from which the conditional reconfiguration in the constructor of ext_i (i from 1 to 4) is false (the interfaces it requires cannot be supplied) but ext_i is notified when the loading of the main window is completed (i.e. after the method *PE.NotifyWindowLoad* an instance of the class ext_i executes). The verification did not find any states that satisfy this CTL formula; this means that the UML models supports this requirement. The source code of each extension contains an *if* statement such that if the method *Main.HasInterface* returns

false the extensions write an error message to the console and return. As a result, the simulation of the UML models and implementation of the DMV agree for this requirement.

- *Each extension executes after the load of the main window is completed if the interfaces the extension requires are supplied.*

  The CTL formula of this requirement is similar to the formula of the previous requirement; however, here we search for states from which the conditional reconfiguration is true (the interfaces the extension requires are supplied) but the extension does not execute after the method *PE.NotifyWindowLoad*. The verification did not find any states that satisfy this CTL formula; this means the UML models supports this requirement. The source code extensions show that if the interfaces are supplied they register to window load events. Thus, the implementation and the simulation of the UML models agree for the this requirement.

- *Extension* 3 *(Ext_3) should not execute the method WindowLoad when the process framework connection is not available.*

  Extension 3 requires the process framework (a gateway to the MRI process framework) to be running so that it can add functionality for reading parameters from the MRI machine. We used the following CTL formula to find states where the method *Ext_3.WindowLoad()* is executed even though the process framework is not available (i.e. the value of the attribute $Ext_3.PFavaliable$ is false):

$$\begin{aligned} &EF(executeMethod(\text{"}ext_3\text{"}.\text{"}ext_3\text{"}) \wedge \\ &(EF(conditionalexecutes(\text{"}false\text{"}.\text{"}PFavaliable\text{"}) \wedge \\ &(EF(returnframe(\text{"}ext_3\text{"}.\text{"}ext_3\text{"}) \wedge \\ &(EF(executeMethod(\text{"}NotifyWindowLoad\text{"},\text{"}PE\text{"}) \wedge \\ &(EF(executeMethod(\text{"}WindowLoad\text{"},\text{"}ext_3\text{"}))))))))))) \end{aligned}$$

  The verification algorithm did not find states that satisfy this CTL formula; thus the requirement is supported by the UML models of the DMV tool. We executed the DMV tool without the process framework and we observed that Extension 3 did not get notified.

- *DMV should support extensions that are dependent on each other; the order in which each extensions method WindowLoad is executed should always be the same.*

  it is expected in the near future that DMV will be extended with extensions that depend on each other. Currently, none of the implemented extensions is dependent on any other. Thus, we modeled two extensions *extV1* and *extV2* where *extV2* is dependent on *extV1*. The reconfiguration requirement for these

extensions is that when both *extV1* and *extV2* are registered as *WindowLoad* observers, then first *extV1.WindowLoad* and, later on, *extV2.WindowLoad* should execute. The simulation with only these two extensions generated a tree with more than 5 branches with 2330 states and 2229 transitions. The following CTL formula verified the existence of an execution sequence where the class *extV2* is notified before *extV1*:

$$\begin{aligned} &EF(executeMethod(\text{"}NotifyWindowLoad\text{"}.\text{"}PE\text{"}) \wedge \\ &\quad (EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}extV2\text{"}) \\ &\wedge (EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}extV1\text{"})))))) \end{aligned}$$

The evaluation shows that the UML models do not support loading of the dependent extensions because the order in which the extensions are notified changes. The sequence diagrams used for this case study show that the class *Decorator* is loaded; however, the implementors did not specify an order during the *dll* files are opened and the class *Decorator* is loaded. The implementation of DMV tool according these models also did not specify a loading order for the extensions and it relies on the ordering of the operating system. As a result, the execution order of the extensions change each time DMV is loaded at a different computer.

- *When there is more than one implementation provider for the same interface, the extensions should be able to execute with each provider.*

  In the current version of the DMV there is only one implementation provider for each interface. However, the design of DMV allows more than one interface provider to be implemented. For example, in near future, the user interface of DMV can change; however, the extensions should be backward compatible in that they should support the old and the new user interface.

  To verify this requirement, we modeled an extension called *UIProvider2* that implements the interface *UI*. This interface contains methods for adding user interface elements to the main window of DMV and the class *UIProvider* implements this interface. By modeling the class *UIProvider2*, we added a second implementation provider for the interface *UI*. The simulation of the modified design generated a tree with more than 120 branches consisting of 22851 states and 22850 transitions. Using the following CTL formula we search for two executions where in the first one an extension runs with *UIProvider* and in the second one, runs with *UIProvider2*:

$$
\begin{aligned}
&(EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}ext_i\text{"}) \wedge \\
&(EF(executeMethod(\text{"}GetFrame\text{"}.\text{"}UIProvider2\text{"}) \\
&\wedge(EF(returnframe(\text{"}WindowLoad\text{"}.\text{"}ext_i\text{"})))))) \\
&\wedge(EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}ext_i\text{"}) \\
&\wedge(EF(executeMethod(\text{"}GetFrame\text{"}.\text{"}UIProvider\text{"}) \\
&\wedge(EF(returnFrame(\text{"}WindowLoad\text{"}.\text{"}ext_i\text{"}))))))))
\end{aligned}
$$

The verification was able to find states that satisfy this formula; thus, the extensions are able to work with either provider and the UML models supports the requirement. We also made a copy of the class *UIProvider* called *UIProvider2* and executed this modified version of the DMV with one extension. We observed that some extensions used the class *UIProvider* and the others used the class *UIProvider2*. So, the requirement is supported by the DMV tool and the verification on the UML models agrees with the implementation.

- *When there is more then one implementation provider for the same interface, all extensions should execute with the same provider.*

  If the simulation generated an execution sequence where Ext_1 executes with *UIProvider2* and Ext_2 executes with *UIProvider* then we conclude that the UML models does not support this requirement. The CTL formula of this execution sequence is:

$$
\begin{aligned}
&EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}ext1\text{"}) \wedge \\
&(EF(executeMethod(\text{"}GetFrame\text{"}.\text{"}UIProvider2\text{"}) \\
&\wedge(EF(returnframe(\text{"}WindowLoad\text{""}ext1\text{"}) \\
&\wedge(EF(executeMethod(\text{"}WindowLoad\text{"}.\text{"}ext2\text{"}) \\
&\wedge(EF(executeMethod(\text{"}GetFrame\text{"}.\text{"}UIProvider\text{"}))))))))
\end{aligned}
$$

  The verification algorithm was able to find states that satisfy this CTL formula. The sequence diagrams of the tool only show that the method *Main.GetImplementors* returns an instance of the implementation provider that implements the interface an extension asks. The case where there is more than one provider is not covered by the UML models. This causes a non-determinism, because which instance the method *Main.GetImplementors* will return is undefined. In the verification of the previous requirement, we showed that this method can in fact return either implementation provider. However, the verification of this requirement showed that the extensions have no control over the providers they receive causing them to

run with different providers. This can cause the tool to crash if, for example, the implementation providers evolve. The problem also existed in the implementation of the tool. As a result, if the designers decide to implement this design, they need to modify the design such that the extensions can ask for specific sub-classes of the interface providers.

4.1.3 Conclusions on the Case Study

With these verifications, we detected that the UML models of the signal viewer tool do not support two of the eight runtime reconfiguration requirements. The designer was not aware of these failures and we provided the designer with the necessary corrections to be made to the design in order to fix the errors. The designer missed these errors because he did not know, for example, that the loading order of the extensions could change. Moreover, we observed that the same errors also occur in the source code of the DMV. For example, the implementation of every extension needs to be changed in order to support the anticipated requirement of having more than one implementation provider (the last requirement described in the previous sub-section). If the verification of this requirement was made before the DMV was implemented, then DMV would be prepared for this anticipated requirement and there would not be any need to change the already implemented extensions. This shows that evaluating the UML models before the implementation is beneficial.

Our conclusions on this case study are the following:

- The result of the verification of the UML models and the manual evaluation by executing the tool; thus, the verification approach works.
- Evaluating the UML models with respect to reconfiguration requirements before the implementation is more beneficial, since costly bug-fix cycles may be avoided.
- The results of the tool are promising.

We also made the following observations during the case study:

- It was hard for the designers to use CTL.
- In case the configuration requirement is not fulfilled by the UML model, a better error report must be generated by the tool.

4.2 Evaluation of the Error Diagnosis Mechanism with CTL

One of the conclusions of the case study with the industry is that the designers/developers would like to see more information when the verification of a reconfiguration requirement fails. For this, we developed an error diagnosis mechanism that shows up-to which point the execution sequence of a reconfiguration requirement is supported (this mechanism is detailed in Section 3.6.1). In order to better grasp the effectiveness of this feedback mechanism we conducted an experiment with computer science students (we did not conduct this experiment with the industry because of the limited human resources the industry can provide).

In our approach, we used ArgoUML as the front end where the users can draw the UML models and convert them to DCMs. Although ArgoUML supports many of the UML's features, the user interface for sequence diagrams contains many errors and it is very hard to use for the unexperienced users. For example, when the user deletes an action accidently, ArgoUML does not allow the user to redraw the same action. We contacted the developers of ArgoUML and submitted bug reports; however, by the time the experiment was conducted we did not get a response. Due to the errors of the user interface, we designed the experiment such that the subjects are allowed to run the tool only once and get the location of the problem. They, then, fix the UML sequence diagrams provided as Visio documents.

This section details the setup, the execution and the results of this experiment. We follow the experiment reporting guidelines presented in the literature [113, 129]. The data analysis is done in SPSS for Windows version 16.0 [11].

4.2.1 Motivation and Overview

The motivation of the experiment is to understand how effective the error diagnosis mechanism based on CTL is. Here, by effectiveness we mean whether using this mechanism helps in finding errors better than not using the mechanism at all. The experiment is conducted with 46 masters computer science students. Before the experiment, the students were given a course on reconfiguration, UML class and sequence diagrams and manually tracing VSL (Section 3.5) specifications. We used the UML models of DMV with two extensions; in total the students have received 6 sequences diagrams and 1 class diagram. To follow the non-disclosure agreements, the names of the classes, attributes, and methods have been changed. The students are also given 3 VSL specifications; one hard, one medium and one easy (decided according to the size of the specification). We injected errors by removing the recon-

figuration mechanisms from the UML sequence diagrams so that all 3 specifications are not supported by the models. For all specifications, the error is located at the last state of the specification. The students were asked to evaluate the specifications and correct only the UML sequence diagrams when the evaluation of the requirement fails. Each specification referred to a distinct set of sequence diagrams. During the experiment, the students are randomly divided into into two groups: one group used the tool that implements the feedback mechanism and the other group manually evaluated the specifications.

4.2.2 Hypothesis

Below is the research question that motivated this experiment:

- Is the feedback mechanism useful for correcting errors related reconfiguration requirements?

Following this question we formulate the following null hypothesis:

- H_0: The tool that implements this feedback mechanism has no effect on the number of errors made by the students.

and the following alternative hypothesis:

- H_{A0}: The tool that implements this feedback mechanism has an effect on the number of errors made by the students.

We choose a significance level of 0.05 for rejecting H_0. It is common practice to aim for the significance level ≤ 0.05; however, for social science experiments a significance level ≤ 0.01 is also a common aim [92].

4.2.3 The Variables of the Experiment

In the experiment we have two units of analysis depending on the presence of the feedback mechanism. Thus, the tool usage is a factor (and the only factor) of this case study. The independent variables are listed below:

1. The UML models: Each student has received the same UML models and the same Visio diagrams. Before the start of the case study, the instructors

Table 4.2: The properties of the VSL specifications used in the case study

Specification Complexity	# of Sequence Diagrams	# of States	# of branches
Easy	1	3	1
Medium	2	6	2
Hard	3	13	4

distributed a paper version of the UML models and loaded the Visio diagrams to the students computer.

2. The size of the specification: Each student received the same 3 VSL specifications. The complexity of the VSL specifications is measured in terms of the number of branches, states, and UML sequence diagrams needed to trace the specification. Table 4.2 provides the categorization of the specifications and the values of the metrics used for the categorization.

3. The time it takes to evaluate the 3 VSL specifications: All subjects are given 3 hours to evaluate all specifications.

4. Injected Error: We injected the same kind of error into the UML models the students received so that all 3 specifications are not supported by the design. All 3 specifications depend on at least one conditional reconfiguration (besides polymorphic reconfiguration and dynamic type loading) and we removed one of the conditional paths from the UML sequence diagrams.

The dependent variable of the case study is the number of errors, which is in absolute scale and counted in the UML models the student has submitted. Since the tool tells whether the VSL specification is supported or not, we only look at the sequence diagrams and count the errors from there. If, for example, out of 3 requirements the student could correct only one, then this student made 2 errors.

4.2.4 Case and Subject Selection

The case is the process of evaluating the 3 reconfiguration requirements on the UML models of the signal viewer tool. The test requirements are inspired by the reconfiguration requirements described in the previous section. We have two groups in this experiment because we have on depended variable with two values: the first group followed the manual evaluation process and the second group received feedback on the location of the problem.

The experiment was integrated into the course on design of software architectures. Hence, the subjects are the 46 MSc computer science students from University of

Table 4.3: The design of the experiment on the effectiveness of the feedback mechanism based on CTL

Factor: Feedback Mechanism	
Level: Present	Level: Not Present
23 Students	23 Students

Twente that attended this course. Manual evaluation of the reconfiguration requirements requires the subjects to be knowledgable about the UML models and Object-Oriented programming. To collect information on how knowledgeable the students are in these two, we asked them the following questions:

1. How many lines of Object-Oriented Code did you program during the last year? The answers are: a) < 100, b) *between 100-5000*, c) *between 5000-10000*, and d) *more then 10000*. 32% of the students selected c, 27% selected d, 18% selected d and 21% selected a.

2. What is your UML expertise? The answers are: a) *used only once*, b) *never modeled in UML*, c) *Used in several course and other porjects*. 72% selected c, 26% selected a and 2% (which is only one student) selected b.

From the answers to these questions, we concluded that the group is knowledgeable about UML and used it at least once, and they have some experience in OO programming.

4.2.5 Experiment Design

We designed the experiment with one factor, which is the presence of the feedback mechanism, and with two levels: feedback mechanism present and not present. Table 4.3 presents the number of participates for each group. The students are randomly assigned to the groups. The group with the feedback mechanism used a version of GROOVE that implements the CTL based feedback. In the rest of this section, we refer to this group as the **tool supported group** and the group without feedback mechanism as the **manual evaluation group**.

4.2.6 Instrumentation

Below are the instruments of the experiment:

- The UML diagrams in Visio and on paper.
- Evaluation log tool.
- VSL specifications on paper.
- The course slides.
- Documents containing the stepwise instructions for students.
- GROOVE with the execution semantics for UML and the extension that implements the feedback mechanism.

4.2.7 Experiment Operation

Before the experiment, the students were given a course on reconfiguration mechanisms, UML, VSL specifications and manually tracing the sequence diagrams to evaluate the reconfiguration diagrams. In this course, the two example reconfiguration diagrams are evaluated on a sample design based on the strategy pattern [59]: one of these requirements was supported by the design and other one was not, so the example also introduced how to make corrections in the sequence diagrams. The slides of this tutorial were uploaded to the course website, so that the students could access it during the experiment.

We prepared stepwise instructions describing what the students should do during the experiment. The instruction states that the students first should redo the examples in the tutorial, so that they are reminded about the manual evaluation process. The step wise instructions included a *reference* section, which provided the semantics of VSL specifications and examples of UML sequence diagrams with reconfiguration mechanisms.

There were two instructors present during the experiment. To circumvent the learning effects, 6 (= 3!) factorial orderings of the specifications were made. As the students entered to the lab room, they received one of the 6 orderings; the instructors recorded the order the student had received. The students were asked to follow the order they received. We also prepared a tool that logged how the student has evaluated the specification. For example, if the student were evaluating the reconfiguration requirement *PlaceHolder* (which is the hard specification; however, the specifications are given names that did not reveal their complexities) first, then in the tool from the first drop down menu the student selected the item *PlaceHolder*. Once this information is provided, the tool logged which UML diagrams the student has opened, and recorded the time the student has started the evaluation. The students were also asked to press the *finish* button when they finished the requirement.

This recorded the time they finished the evaluation. Due to many computer failures due to operating system updates the time information became unreliable, so it is not used in the data analysis.

The tool supported group was taken to a different lab room, where one of the instructors explained them how the tool was used and what information they could expect from the tool. The log tool was also used in the tool supported group.

The students were asked to make the corrections in the Visio diagrams and send the Visio diagrams and the output of log tool with an email to the instructor when there finished the evaluation of 3 VSL specifications. The students were also given paper versions of the UML models to ease the tracing process; these papers were also collected at the end of the session.

At the end of the session, the students were also given a survey that asked how confident the student was about her/his results and, if the student was in the tool group, at what level the tool did help and how the tool could be improved. The surveys were also collected by the instructors.

Before the actual experiment, we made a trail run with two subjects to ensure that the instructions and UML diagrams were clear.

4.2.8 Data Analysis

From the received emails, we discovered that two Visio files were corrupted; thus, we could not count the number of errors for these two students. These students were from the tool supported group. Figure 4.2.8 presents the descriptive statistics of the experiment data; the raw data for this experiment can be found in Section B.1. Here, it can be seen that the minimum number of errors in both groups is 0; however, in the manual evaluation group there is one student who made no errors whereas in the tool supported group there are 4 students. Although for each group there were students who made an error in all 3 specifications, in the tool supported group 2 students made an error in all specifications whereas in manual evaluation group this number is 6. The mean number of errors in the tool supported group is 1.38 whereas in the manual evaluation group the mean is 1.95. In each group, mostly the students failed to correct the hard specification. Only one student from the manual evaluation group was able to correct this specification and from the tool supported group 6 students were able to correct it.

Comparing the Kurtosis values for both groups, it can be concluded that the distribution of the number of errors in the manual evaluation group is relatively flat, while in the tool supported group the samples are distributed around the mean. The

	Group			Statistic	Std. Error
Errors	Manual	Mean		1,9565	,20351
		95% Confidence Interval for Mean	Lower Bound	1,5345	
			Upper Bound	2,3786	
		5% Trimmed Mean		2,0000	
		Median		2,0000	
		Variance		,953	
		Std. Deviation		,97600	
		Minimum		,00	
		Maximum		3,00	
		Range		3,00	
		Interquartile Range		2,00	
		Skewness		-,228	,481
		Kurtosis		-1,349	,935
	Tool	Mean		1,3810	,17561
		95% Confidence Interval for Mean	Lower Bound	1,0146	
			Upper Bound	1,7473	
		5% Trimmed Mean		1,3651	
		Median		1,0000	
		Variance		,648	
		Std. Deviation		,80475	
		Minimum		,00	
		Maximum		3,00	
		Range		3,00	
		Interquartile Range		1,00	
		Skewness		1,064	,501
		Kurtosis		,515	,972

Figure 4.3: The descriptive statistics of the experimental data. The data consists of the analysis made on the number of errors made without tool and with tool.

negative skewness of the manual evaluation group means that most of the samples have high values; for example, there is one student that made no errors. In the tool supported group, the positive skewness indicates that most samples have lower values. Most of the students in the tool supported group made one error, mostly in the hard specification.

The Kolmogorov-Smimov normality analysis yielded a significance less then 0.01 for both groups. From this we cannot conclude that the underlying distribution for these are not normal. The histograms of the data presented in Figure 4.2.8 shows that the distribution of the manual evaluation group is linear and the the distribution of the tool supported group is reasonably normal. For the rest of our analysis, we assume that distributions are normal (and take this assumption as a validity threat).

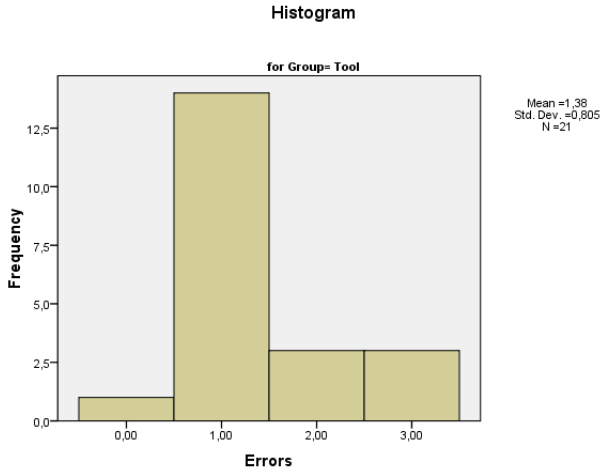
Hypothesis Testing

For testing the hypothesis, we used the independent samples t-tests. The assumptions of this test are independence of observations, normal distribution for the test variable and random samples. The independence of the observations is achieved by executing the experiment for the two treatments at the same time. The random samples assumption of this test is achieved by randomly distributing the students to the groups.

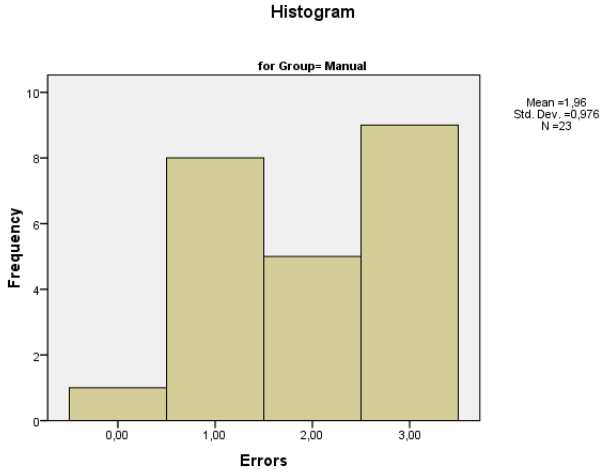
Figure 4.2.8 presents the results of the independent values analysis. The t-test table has two rows: the first row contains the results of the t-tests when the variances are assumed to be equal and the second row shows the results when the variances are assumed to be not equal. The significance value for Levene's equality test for equality of variances is 0.174, this value is greater then the accepted threshold value 0.05 so the variances for both groups are said be equal and we focus on the first row of the t-test. Here, the two tailed significance value is 0.04. This value is less then the aimed significance value of 0.05. From this, we can conclude that the number of errors made by the tool supported group (M=1.38 Std.=0.8) is significantly different than the number of errors made by the manual evaluation group (M=1.96 Std.=0.98, t(42)=2.122 p=0.05) so H_0 can be rejected.

Survey Results

The students were also given a survey when they finished evaluating the 3 VSL specifications. The survey asked the students their confidence on their answers and for tool supported group at what point the tool did help them. The question about



(a)



(b)

Figure 4.4: The histogram of the number of errors made by (a) the tool supported group and (b) the manual evaluation group

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
Errors	Equal variances assumed	1,913	,174	-2,122	42	,040	-,57557	,27120	-1,12287	-,02827
	Equal variances not assumed			-2,141	41,593	,038	-,57557	,26880	-1,11819	-,03295

Figure 4.5: The results of the independent samples t-test for assessing the differences between the tool supported and the manual evaluation group.

Table 4.4: The students' answers to the question at what point the tool has helped them

Answer	% of Students
The tool did not help at all	9%
The supported/not supported answers were useful	20%
The point of failure was useful	61%
Other	9%

confidence asked the students to rank their confidence on their answers from 5 (very confident) to 0 (not confident at all). Interestingly, the average confidence levels of both groups were very close: the average confidence level of the tool supported group was 3.18 and the average confidence level of the manual evaluation group was 3.14. This shows us that the tool (based on CTL error diagnosis) was not specifically pointing out the problems; so, the students had to do some manual tracing to find the problem.

The results of the students' answers to the question at what point the tool did help you is presented in table 4.4. Here, we see that most of the students found the feedback on the point of failure useful. The tool also reported whether the requirement has failed or not and 20% of the students found only this part to be useful. The students who selected *Other* stated that they also found this answer useful but included that the point of failure reported by the tool is actually not the real point of failure, which is confusing. From these results, we can say that 38% of the students did not find the feedback mechanism useful.

The tool supported group was also asked in which ways the tool could be improved. The majority said that the tool should be integrated with an UML tool. The students also suggested that the tool would be more effective if it displayed the complete trace of the execution to the failure point rather then just stating the point.

4.2.9 Validity Evaluation

This section discusses the threats to the validity of the experiment. The discussion follows the main validity threats provided in [129]. For each validity threat, we provided a brief definition about the threat; however, interested readers are referred to the literature for detailed descriptions [129].

Table 4.5: The inputs to the G*Power tool and the $1 - \beta$ power of the experiment

Input	H_0
μ_{Manual}	1.96
μ_{Tool}	1.38
σ_{Manual}	0.98
σ_{Tool}	0.8
n_{Manual}	23
n_{Tool}	21
α	0.05
$1 - \beta$	0.68

Conclusion Validity

The threats to conclusion validity are issues that affect the ability to make conclusions on the data analysis. For this experiment, we identified two threats that should be detailed:

- **Statistical power**: The statistical power is the ability of the test to reveal a true pattern in the data. If the power is low then there is high chance that an erroneous conclusion is drawn about the data. To test the power, we used post-hoc analysis with the G*Power tool [53]. Table 4.5 lists the input provided to the G*Power tool; the α is the significance value we aimed at for the experiment. The power analysis yielded a power of 0.68; this value is lower than the accepted power value of 0.8. This shows that although the experiment's power is low, the conclusions can still be accepted, but the conclusions would possibly be more powerful with more subjects. The normality analysis also agrees with the findings of the power test. From this we conclude that there is room for improvement in the experiment to draw better conclusions.

- **Reliability of treatment implementation**: This threat arises when there are differences in treatment implementation; the implementation of the treatment should be as standard as possible for each group. To prevent the manual evaluation group to learn about the feedback mechanism, the subjects have entered to the experiment at the same time. This required different instructors to give the instructions. Thus, there may be a threat to the reliability of treatment implementation.

Internal Validity

The threats to the internal validity are issues that can effect the measurements of the independent variable, without the researchers knowledge. We identified 3 threats that can effect the internal validity of the experiment. Below, we describe the precautions we took before and during the experiment to address these validities:

- **Maturation**: this threat occurs when the subjects get bored (negative) or have unintended learning (positive). To prevent unintended learning, the instructions gave an example and the course slides also contained the examples made during the course. The example in the instructions was the first assignment the students made.

  In order to prevent the students to learn about the feedback mechanism (or the tool support) the students were not allowed to go outside the rooms where the experiment took place (two students left for a short break with permission from the instructor). This may introduce a negative maturation threat because the students may be bored as they progress in the experiment.

- **Instrumentation**: The instrumentation threat arises from improper preparation of documents, data collection forms etc. We conducted a preliminary run of the experiment with two subjects to see if the instructions contained errors or were misleading. These participants were others than the participants of the actual run of the experiment and the data from these participants are not included in the data collected at the actual case study.

  The experiment was also designed to include the time it takes the students to finish corrections in a VSL specification. However, during the experiment several computers have crashed and the time data also included the time it takes to restart the computer. Because of this the time data became unreliable, so we discarded them in the data analysis.

  At the beginning of the experiment, a student spotted a typo in one of the sequence diagrams. The instructors then let the both classes know about the situation and the correction.

- **Diffusion or Imitation of Treatments**: This threat arises when the subjects learn about the treatments of the experiment. To circumvent this threat, students did not know the existence of the tool. The experiment is conducted for each group at the same time and the students at the manual evaluation group learned about the tool at the end of the experiment.

UML Experience	Tool	Manual
Never used it before	0	1
Used it only once	5	6
Used it several times	18	16

Table 4.6: The UML experience of students for the tool and the non-tool groups

Construct Validity

The construct validity is the ability to draw conclusions about the relation between the experiment and the hypothesis. The threats to this validity affect the correctness of this relation. We identified two types of threats that effect this relation:

- **Confounding constructs and the level of constructs**: These threats arise when there are confounding constructs that are not taken into account for the experiment. The experience of the students is a confounding construct; the students are randomly divided into the groups and thus the experience level of students can be unbalanced between groups. Table 4.6 shows the distribution of the students and the answer they have given to the question about their UML experience. This table shows that student that has never modeled in UML was in the manual evaluation group. However, the average experience levels of the groups are close (experience level of 1.65 for manual group and experience level of 1.78 for tool supported group), so we think that this threat did not effect the results.
- **Experimenter Expectancy**: The experimenters may bias the result of an experiment because they may have expectations about the results. As the experimenter, the author of this thesis expected that the feedback mechanism would helped the students. However, the experiment is designed and the data is analyzed with experts (other than the promoter and the co-promoter of this thesis) that have no expectations about the outcome of the experiment.

External Validity

Threats to the external validity limit the ability to generalize the results of the experiment. The following threats may limit the external validity of the experiment:

- **Interaction of Selection and Treatment**: This threat arises when the subjects are not representative of the population. The survey results shows that the students have knowledge on UML and OO programming. This may not

represent the general population of developers/designers. However, the feedback mechanism and reconfiguration requirement evaluation on UML models assumes the developers/designers to have some knowledge about OO programming and UML. We think that the students who attended the experiment are a good representative of a population that has knowledge in both of these subjects. Thus, the results of the experiment can be generalized to developers/designers working with UML and OO.

- **Interaction of Setting and Treatment**: This threat arises when the experimental setting or the tools are not representative. We used the UML models of the signal viewer tool which is an industrial tool. The medium and the hard requirement are very similar to the last two requirements presented in the previous section. The error for the hard requirement (which dealt with the loading of dependent extensions) was also very similar to the actual error. Thus these are representative of the industrial setting. However, the easy requirement (which almost everyone corrected correctly) and the errors for the requirement medium and easy are not that representative. For example, for these two requirements the error was not dependent on the execution up to the point where the requirement had failed. We could not use the actual errors because they were dependent on the environment in which the tool works, which is very domain oriented (i.e. they require some knowledge about the MRI framework). Thus, there may be a threat to generalize these results to industry.

4.2.10 Conclusions on the Error Diagnosis Mechanism

Although the hypothesis testing showed that we can reject the null hypothesis, we find the mean number of errors 1.34 to be much higher than our expectations. This can be because of the feedback mechanism is not effective enough, because the students were able to use tool only once or because the hard specification was very hard that students could not do it. The former two cases is backed up by the survey results and the later case is backed up by the fact that majority of the students made an error in the specification. Thus, our conclusions on this experiment are the following:

- The error diagnosis mechanism provides some beneficial guidelines on the location of the problem.
- There is room for improvement in the error diagnosis mechanism based on CTL.

- The experiment should be repeated with:
  1. More balanced VSL specifications.
  2. More subjects.
  3. Feedback mechanism integrated with UML tools.

4.3 Evaluation of the Error Diagnosis Mechanism with Control Automata

We conducted another experiment to test the effectiveness of the feedback mechanism based on a control automaton. The design of this experiment is very similar to the experiment described in the previous section and the subjects are also computer science master students. This experiment is integrated to with a different course and we ensured that students that did the previous experiment were not included in this experiment.

In the previous experiment, the majority of the students wanted the tool to be integrated with UML tools. Because of the problems with open source UML tools (and XMI libraries), we defined a textual syntax for UML sequence and class diagrams using Textual Concrete Syntax (TCS) [75]. For a given meta-model (expressed in Kernel Meta Model (KM3) [74]) and syntax to textually express a model in this meta-model, TCS automatically generates model-to-text and text-to-model tools. We expressed a meta-model for UML class and sequence diagrams in km3 and defined a textual syntax for these two models (the details of the syntax are presented in Appendix C). Then, we generated a text-to-model converter using TCS; the converter generates an ECORE model and writes the model into an XMI file. We programmed an ECORE to GXL converter and integrated it to GROOVE. In this way, the generated XMI files can be loaded from GROOVE without launching the converter separately.

This section reports on the execution and the results of this experiment. We follow the style presented in the previous section and we conducted the data analysis again in SPSS [11].

4.3.1 Motivation and Overview

The motivation of the experiment is the understand how effective the error diagnosis mechanism based on a control automaton. Similar to the previous experiment, we

want to know whether using this error diagnosis mechanism helps in correcting reconfiguration errors or not. The experiment is conducted with 22 master computer science students; these students did not participate in the previous experiment. Before the experiment the students were given a course on reconfiguration, UML class, sequence diagrams, the textual form of these diagrams and manually tracing VSL specifications. We used the UML models of the signal viewer tool with three extensions; in total the students have received 4 sequence diagrams and 1 class diagram. To follow the non-disclosure agreements, the names of the classes, attributes, and methods have been changed.

The control automaton based feedback mechanism can provide feedback for reconfiguration requirements that search for the existence of the reconfiguration (requirement type 1) and the invariant (requirement type 2). Thus, the effectiveness of the mechanism should be tested for both type of requirements. We achieved this by conducting two experiments at the same time: Experiment 1 (E_1) tested the effectiveness of the tool for first type of requirements and Experiment 2 (E_2) tested the effectiveness of the tool for requirements of type 2. The students were given 4 VSL specifications; two specifications of requirements of type 1 and two specifications of requirements of type 2. The students were divided into two groups; the first group did all the specifications about requirements of type 1 manually and the specifications of requirements of type 2 with tool support. The second group evaluated the specifications of requirement of type 1 with tool support and the specifications of requirements of type 2 manually. For each requirements type, the students have received one easy and one hard specification. We injected errors by removing the reconfiguration mechanisms from the UML sequence diagrams so that all 4 specifications were not supported by the models. For all specifications, the error is located at the last state of the specification. The students were asked to evaluate the specifications and to correct only the UML sequence diagrams when the evaluation of the requirement fails. Each specification referred to a distinct set of sequence diagrams and a distinct file that has the textual format of the UML models.

4.3.2 Hypotheses

Below is the research questions that motivated this case study:

- Is the feedback mechanism useful for correcting errors related reconfiguration requirements?

Following this question we formulate the following null hypothesis for E_1:

- H_{E1}: The tool that implements this feedback mechanism has no effect on the number of errors made by the students for the reconfiguration requirements of type 1.

and the following alternative hypothesis:

- H_{AE1}: The tool that implements this feedback mechanism has an effect on the number of errors made by the students for the reconfiguration requirements of type 1.

For E_2 the following null hypothesis is assumed:

- H_{E2}: The tool that implements this feedback mechanism has no effect on the number of errors made by the students for the reconfiguration requirements of type 2.

and the following alternative hypothesis:

- H_{AE2}: The tool that implements this feedback mechanism has an effect on the number of errors made by the students for the reconfiguration requirements of type 2.

We aim for a significance level of 0.01 for rejecting these hypotheses. This significance level is lower than the level we aimed for the previous experiment because for this experiment the evaluation process is integrated to a UML tool and, thus, we expect the number of errors made with tool support to be much more significantly different than the number of errors made without tool support.

4.3.3 The Variables of the Experiment

The factor of both E_1 and E_2 is the presence of the tool that implements the feedback mechanism. The independent variables are listed below:

1. The UML models: Each student has received the same UML models and the same Visio diagrams. Before the start of the case study, the instructors distributed a paper version of the UML models and loaded the Visio diagrams to the students computer.

Table 4.7: The properties of the VSL specifications used in the case study

Requirement type	Specification Complexity	# of States	# of Sequence Diagrams required for tracing
Requirement Type 1	Easy Hard	3 6	1 3
Requirement Type 2	Easy Hard	3 6	1 3

2. The size of the specification: Each student received the same 4 VSL specifications (in different order). The complexity of the VSL specifications is measured in terms of the number of states, and UML sequence diagrams needed to trace the specification. Table 4.7 provides the categorization of the specifications and the values of the metrics used for the categorization. It is important to note that all the specification required different sequence diagrams; to trace specification 1 the students had to load UML Sequence diagram 1.

3. The time it takes to evaluate the 4 VSL specifications: All subjects are given 3 hours to evaluate all specifications.

4. Injected Error: We injected the same kind of error to the UML models the students received, so in total 4 errors are inject and all 4 specifications are not supported by the design. All 4 specifications depend on at least one polymorphic reconfiguration (besides conditional reconfiguration and dynamic type loading) and we removed the *PolymorphicReconfiguration* tag from the call action. The specifications are independent of each other; fixing the error for a specification, for example, has no effect on the other specification

The dependent variable of the case study is the number of errors which is in absolute scale and counted in the UML models the student has submitted. Since the tool tells whether the VSL specification is supported or not, we only look at the sequence diagrams and count the errors from there. If, for example, for requirement type 1 out of 2 requirements the student could correct only one, then this student made 1 error.

4.3.4 Case and Participants

The case is the process of evaluating the 4 reconfiguration requirements on the UML models of the signal viewer tool; 2 reconfiguration requirements are of type

Table 4.8: The design of the experiments E_1 and E_2

Factor: The use of the tool	
Level: Present	Level: Not Present
9 Students	10 Students

1 and 2 reconfiguration requirements are of type 2. The requirements are similar to the actual requirement of the signal viewer tool when loading two dependent extensions. For each experiment, we have two groups: the first group followed the manual evaluation process and the second group used GROOVE with the feedback mechanism based on control automaton.

The experiment was integrated to the course on Advanced Design of Software Architectures: Model Driven Engineering. Hence, the subjects are the 19 MSc computer science students from University of Twente that attended this course. Manual evaluation of the reconfiguration requirements requires the subjects to be knowledgable about the UML models and Object-Oriented programming. To collect information on how knowledgeable the students are in these two subjects, we asked them the following questions:

1. How many lines of Object-Oriented Code did you program during the last year? The answers are: a) < 100, b) *between 100-5000*, c) *between 5000-10000*, and d) *more then 10000*. 45% of the students selected c, 20% selected d, 35% selected d and 0% selected a.

2. What is UML expertise? The answers are: a) *used only once*, b) *never modeled in UML*, c) *Used in several courses and other projects*. 90% selected c, 0% selected b and 10% selected a.

From the answers to these questions, we concluded that the group is knowledgeable about UML (used it at least once) and they have some experience in OO programming.

4.3.5 Experiment Design

We designed each experiment with one factor, which is the presence of the feedback mechanism, and with two levels: feedback mechanism present and not present. Table 4.8 presents the number of participates for each group. The students are randomly assigned to the two groups: one group (group 1) performed E_1 with tool and E_2 manually, and the other group (group 2) performed E_1 manually and E_2

with the tool. According to this arrangement each student used the tool; however, to evaluate different kinds of requirements. Because the output of the mechanism differs for different types of requirements, with this allocation of the students to the groups allows us to achieve independence of observations. That is, in this allocation for E_1 the **manual evaluation group** was group 2 and the **tool supported group** was group 1 and for E_2 the manual evaluation group was group 1 and the tool supported group was group 2.

4.3.6 Instrumentation

Below are the instruments of the experiment:

- The UML diagrams on paper.
- Eclipse with TCS and an eclipse project containing the UML diagrams in textual format.
- VSL specifications on paper.
- The course slides.
- Documents containing the stepwise instructions for students.
- GROOVE with the executions semantics for UML and the extension that implements the feedback mechanism.

4.3.7 Experiment Operation

Before the experiment the students were given a course on reconfiguration mechanisms, UML, the textual syntax used for UML sequence and class diagrams, VSL specifications and manually tracing the sequence diagrams to evaluate the reconfiguration diagrams. In this course, all students have also received instructions on how to use the tool. We prepared stepwise instructions describing what the students should do during the experiment. Before evaluating any specifications the students are given two examples to familiarize themselves with the textual form of the UML diagrams. The first example specification is supported by the design and instructions for this specification showed the students how to find the textual version of classes, classifiers and call actions in the textual representation of the call action. The second examples specification was not supported by the design. For this example, first the students had manually trace the diagrams and find the answer; no

instructions on how to manually trace the UML diagrams and the textual format of the diagrams was present. After manually tracing, the students were presented with the instructions on how to convert the textual format UML diagrams to XMI and load this XMI file from GROOVE. Then, the students were asked to redo the example with the tool.

There were three instructors present during the experiment. To circumvent the learning effects, 24 (= 4!) orderings of the specifications were made. As the students entered to the lab room they received one of the 24 orderings; the instructors recorded the order the student had received. The students were asked to follow the order they received. After evaluating the examples, the students were handed out a *map* sheet describing which UML sequence diagram and textual version of the UML diagrams are required for each specification. For example, the specification *PlaceHolder* requires the sequence diagram named *PlaceHolder* and textual file *PlaceHolder.sd*. The map sheet also showed whether they should evaluate a specification manually or not. The electronic versions of the specifications were not handed to the students, to ensure that they used the tool when they supposed to. If a student's map stated that the specification *PlaceHolder* should be evaluated with the tool, the student asked the instructor to load the specification. Each instructor is given a USB stick containing the specifications. Instructions on how to load the specifications to GROOVE are provided to the instructors before the experiment.

The students were asked to make the corrections to the textual format of the UML diagrams and email the corrected files to the instructor if they found out that the UML diagrams does not support the specification. If they found that the specification were supported then the students were asked to send an email stating that the specification is supported with the name of the specification. The students were also given paper versions of the UML diagrams for convenience.

At the end of the session, the students were also given a survey that asked how confident the student is about the her/his results and if the student were in the tool group, at what level the tool has helped and how the tool can be improved.

Before the actual experiment, we made a trial run with two subjects to ensure that the instructions and UML diagrams were clear.

4.3.8 Data Analysis

The data analysis is done by counting the number of errors in the files the students have sent through email. Two students forgot to attach some of the files, so their results are not included in the data analysis. We also discovered that one of the

	GroupE1			Statistic	Std. Error
ErrorsE1	Manual	Mean		1,2500	,25000
		95% Confidence Interval for Mean	Lower Bound	,6588	
			Upper Bound	1,8412	
		5% Trimmed Mean		1,2778	
		Median		1,0000	
		Variance		,500	
		Std. Deviation		,70711	
		Minimum		,00	
		Maximum		2,00	
		Range		2,00	
		Interquartile Range		1,00	
		Skewness		-,404	,752
		Kurtosis		-,229	1,481

Figure 4.6: Descriptive statics for E1, the tool supported group did not make any errors so the results of this group is omitted.

instructors has selected the wrong requirement type during one students evaluation (the student pasted the output of the tool). The data for this student is also discarded from the analysis. Thus, the data analysis is conducted over 8 students per group (in total 16 students). The tool supported group in both experiments made 0 errors; that is made the right corrections to the design. This means that the tool supported group continued evaluation until the tool reported that the requirement is supported. Figure 4.6 presents the descriptive statistics for E_1; the raw data for this experiment can be found in Section B.2.1. Because the number of errors for the tool supported group is 0, we do not provide the descriptive statistics for this group.

The mean number of errors in the tool supported group is 0, whereas in manual group the mean number of errors is 1.25. 3 students have made errors all specifications and only one student has made no errors in the manual group. The 5% trimmed mean is very close to the actual mean of data; this means that the extreme values do not have a strong influence on the original mean. The negative skewness in the manual group suggest that most of the data are greater than the mean. The negative Kurtosis value suggests that the data is peaked and clustered around the mean. The Kolmogorov-Smimov normality test yielded a significance of 0.109. This value is larger then 0.01, which suggests the distribution of the data is normal.

Figure 4.7 presents the descriptive statistics for E_2; because the the number of errors for all students in the tool supported group is 0, we omit the descriptive statistics for the tool group (the raw data for this experiment can be found in Section B.2.2). Here, the mean number of errors in the manual evaluation group is 1.12 whereas this number is 0 in the tool supported group. Two students from the manual group have made no errors and three students have made an error in all specifications. The 5% trimmed mean (1.14) is very close to the actual mean of the data suggesting that the extreme values do not have a strong influence on the actual mean. The negative

	GroupE2			Statistic	Std. Error
ErrorsE2	Manual	Mean		1,1250	,29505
		95% Confidence Interval for Mean	Lower Bound	,4273	
			Upper Bound	1,8227	
		5% Trimmed Mean		1,1389	
		Median		1,0000	
		Variance		,696	
		Std. Deviation		,83452	
		Minimum		,00	
		Maximum		2,00	
		Range		2,00	
		Interquartile Range		1,75	
		Skewness		-,277	,752
		Kurtosis		-1,392	1,481

Figure 4.7: Descriptive statics for E2, the tool supported group did not make any errors so the results of this group is omitted.

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
ErrorsE1	Equal variances assumed	18,290	,001	-5,000	14	,000	-1,25000	,25000	-1,78620	-,71380
	Equal variances not assumed			-5,000	7,000	,002	-1,25000	,25000	-1,84116	-,65884

Figure 4.8: Independent values t-test analysis for E_1

skewness indicates that most students have made more than 1 error. The negative Kurtosis value suggests that the data is peaked and clustered around the mean; this Kurtosis value of is lower then the Kurtosis value of E_1, which means that the data peaks shaper around the mean for E_2. The Kolmogorov-Smimov normality test yielded a significance value of 0.2, which suggests that the underlying distribution is normal.

4.3.9 Hypothesis Testing

We used the independent values t-test for testing H_{E1}. The assumptions of the t-test hold in E_1: the dependent variable "number of errors" is measured in ratio scale; each participant is assigned to a group and an experiment randomly; the experiment for each group is conducted at the same time, so the observations are independent of each other; the Kolmogorov-Smimov value of the number of errors is greater than 0.01 so the underlying distribution is likely to be normal.

Figure 4.8 presents the results of the independent values t-test. The Levene's test for equal variances resulted in a significance value of 0.001 which is less than 0.05 so the equality of variances cannot be assumed for these values. Because of this, we

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
ErrorsE2	Equal variances assumed	16,869	,001	-3,813	14	,002	-1,12500	,29505	-1,75782	-,49218
	Equal variances not assumed			-3,813	7,000	,007	-1,12500	,29505	-1,82268	-,42732

Figure 4.9: Independent values t-test analysis for E_2

focus on the second row in the t-test table. Here the significance value is 0.002. We aim for a p-value of 0.01 for the experiment. The significance value of t-test is lower then the aimed p-value, so there is a significant difference in the means of the two groups. Thus, we can conclude the number of errors made by the tool supported group (Mean = 0, Std. Dev.= 0) significantly different than the number of errors made by the manual evaluation group (Mean= 1.24, Std. Dev.=0.7, t(10) = 5, p = 0.01) and H_{E1} can be rejected.

For testing H_{E2}, we also used the independent values t-test. The assumptions of the t-test hold in E_2: the dependent variable "number of errors" is measured in ratio scale; each participant is assigned to a group and an experiment randomly; the experiment for each group is conducted at the same time so the observations are independent of each other; the Kolmogorov-Smimov value of the number of errors is greater than 0.01 so the underlying distribution likely to be normal.

Figure 4.9 presents the results of the independent values t-test. The Levene's test for equal variances resulted in a significance value of 0.001 which is less than the threshold significance value of 0.05 so the equality of variances cannot be assumed for these values. Because of this, we focus on the second row in the t-test table. Here, the significance value (p-value) is 0.007 which is less than the aimed p-value of 0.01. Thus, the number of errors made by the tool supported group (Mean = 0, Std. Dev.= 0) is significantly different than that of the manual evaluation group (Mean= 1.13, Std. Dev.=0.83, t(10) = -3.18, p = 0.01) and H_{E2} can be rejected.

4.3.10 Survey Results

To asses how the tool has helped the students, the students were given a survey at the end of the session. Table 4.9 shows that 75% of the students found the trace provided by the tool useful; however, 25% said that they did not use the trace. From this 25% of the students the majority said that they wanted a better user interface for the tool.

We also asked the students to rank their reaction on the statement "I found the

Table 4.9: The students' answers to the question at what point the tool has helped them

Answer	% of Students
The tool did not help at all	10%
The supported/not supported answers were useful	15%
The trace provided was useful	75%

textual version of the UML diagrams hard to follow" from *1* (strongly disagree) to *5* (strongly agree). 25% of the students said 1, 50% said 2, 15% said 3, 10% said 4 and 0% said 5. From this, we conclude that the students found the language easy to follow.

4.3.11 Validity Evaluation

This section describes the validity evaluation of the experiment. We analyze the validity the following three main points (for definitions of the validity threats the reader is referred to Section 4.2.9):

Conclusion Validity

We identified the following threats to derive conclusions on the results of the both experiments:

- **Statistical Power:** We used the G*Power [53] tool to conduct a post-hoc power analysis for the statistical power of the experiment. Table 4.10 presents the input and the output $1 - \beta$ of G*Power. Here, G*Power did not allow to enter the 0 as standard deviation, thus we approximated this value by entering 0.00001 as standard deviation. With these values, the power of each experiment $(1 - \beta)$ is greater then the accepted low power value 0.8. Since the power of both experiments are greater than the low power value, the statistical powers of the experiments are significant and low power is not a threat.

  The standard deviation approximation can be considered as a threat to the validity to draw conclusions from the rejection of H_{E1} and H_{E2}. However, this threat does not arise because of an error in the design of the experiment. It is an outcome of the experiment; the tool supported group continued on correcting a specification until the tool has reporters the requirement is supported by the tool.

Table 4.10: The inputs to the G*Power tool

Input	H_{E1}	H_{E2}
μ_{Manual}	1.25	1.13
μ_{Tool}	0	0
σ_{Manual}	0.7	0.83
σ_{Tool}	0.00001	0.00001
n_{Manual}	8	8
n_{Tool}	8	8
$1-\beta$	0.98	0.87

- **Reliability of Treatment Implementation**: During the experiment, the Eclipse environment in 2 computers has crashed due to the way the students have corrected the specification. Due to this crash, the students could not use the tool to recheck whether the diagrams they corrected supports to the requirement or not. The error occurred in the easy specification of the E_2 and for the easy specifications finding and correcting the errors with the output of the tool is very straight forward. So, the two students were able make the right corrections such that the requirements are supported. However, because they could not reuse the tool to check their changes to the diagrams for these 2 students the treatment was not implemented as the rest of the students. Because of this, there may be a threat to the reliability of treatment implementation.

Internal Validity

The following are identified as the threats to the internal validity of the data:

- **Maturation**: The students are given two example specifications to reduce the effects of unintended learning. The examples are the first specifications which the students have worked on.
- **Instrumentation**: The Eclipse environment in 2 students computer crashed during the evaluation of the requirement easy for experiment 2. The way the students have corrected the specification resulted in a null pointer exception. Because of this, these students could not retest their corrections. However, the specification was very easy and the students corrected the error. These students are included in the data analysis.
- **Diffusion or Imitation of Treatments**: All students were given a chance to use the tool, so during the experiment they learned about the treatment. The

electronic version of the specifications was not handed to students to prevent them from using the tool for specifications that they should evaluate manually. As a result, the influence of the tool is limited to the specifications for which the students should use the tool.

Construct Validity

For both experiments, we identified and addressed the following threats in the design process:

- **Experimenter Expectancy**: The experiment is conducted to provide empirical evidence on how the tools perform. As the developer of the approach, the author of this thesis thinks the tools are beneficial. However, we collaborated with experts (other than the promoter and the co-promoter of this thesis), who does not have specific expectations about the experiment, in designing the experiment.
- **Interaction of different treatments**: This threat arises when the subjects are involved in more then one study, treatments from different studies may interact. In these experiments, students are subject to different treatments because the output of the tool and the evaluation process for requirement types 1 and 2 differ. Because the type of the correction does not change, the students get experienced as they progress in the experiments. This may cause, for example, a student to do E_2 better than E_1. To circumvent this threat, we randomized the order of the experiments and specifications. Thus, some students did E_2 before E_1 or they received the specifications of both experiments interleaved (e.g. E_1 hard, E_2 easy, E_1 easy, E_2 hard).
- **Confounding Constructs and Levels of Construct**: During the experiment, the students were randomly divided to groups. This may cause, for example, experienced students to be assigned to the tool support group. The survey results show that the students knowledge in UML and OO programming is reasonably homogenous. However, there are still some students with less knowledge on these subjects, which may be a threat to the results.

External Validity

The external validity includes threats to the generalization of the results of the experiment. From these threats, we identified the following two as important to both experiments:

- **Interaction of Selection and Treatment**: We aimed to test the effectiveness on a population that is knowledgeable about UML and OO programming. Due to this, the experiments cannot be generalized to for a large group of students or developers. This, however, does not devaluate the results of the experiment because we wanted to have empirical evidence that the tools are useful in a homogenous set of students that are knowledgeable in the subjects required by the experiment.

- **Interaction of Setting and Treatment**: Both experiments used the UML diagrams of an industrial tool. The reconfigurations requirements are inspired by the real reconfiguration requirements presented in Section 4.1. Compared to the reconfiguration errors we discovered in the DMV, the errors we injected to the diagrams are relatively simple. For example, an error effects only one extension (i.e. the effect on the design is local). This may be a threat to generalization of the results to industry. We choose to inject simple errors because identifying errors that expand to more than one extension requires expertise on the design which students do not have.

4.3.12 Conclusions on the Error Diagnosis Mechanism

For both experiments, the hypothesis testing and the descriptive statistics showed that the tool indeed helped the students. The powers of both experiments are still higher then the threshold 0.8; so the conclusion of rejecting the null hypothesis in both experiments is powerful. Below we list our conclusions on the experiment:

- The feedback mechanism is useful in error diagnosis.
- The user interface of the tool should be improved.
- The current results are conclusive.

4.4 Conclusion and Future Work

We conducted a case study and two experiments on the process for computer aided reconfiguration requirements verification. To make the experiments close to real life, the UML models of an industrial software were used. For the case study and the experiments, the tools helped the designer and the students in the evaluation process. Thus, our conclusion is that our approach is suitable for helping the designers and

the requirements engineers to verify the runtime reconfiguration requirements of UML models.

From the case study with the industry, we observed that CTL is hard for designers to use and the designers needed more insight on when the verification of a reconfiguration requirement fails (e.g. that evaluates the CTL formula to false). We developed the VSL for specifying the execution sequences that conform to reconfiguration requirements. This language used the elements from UML activity/state diagrams. We also developed two error diagnosis mechanisms that provide feedback on the possible location of the problem to the designers: one based on CTL and the other based on a control automaton. The experiment conducted to test the effectiveness of the mechanism based on CTL showed that the mechanism is indeed helpful. However, the data analysis and student surveys showed that using the tool only once to get feedback on the location of the problem is limiting and there is room for improvement of the tool. Because of this, we decided to improve the feedback mechanism and repeat the experiment.

The experiments conducted on the effectiveness of feedback mechanism based on a control automaton, showed very promising results. For example, the students used the tool until the tool reported that the requirement is supported. As a result, the tool supported group did not make any errors. The students in the manual evaluation group on the other hand made at least one error. The statistical power of these experiments was high which enforces that hypothesis tests. As a future work, we plan to develop extensions to industry accepted UML tools (such as Borland Together [3] and Rational Rose [10]) so that the process and the feedback mechanisms can be used during the development of industrial software. In this way, we can conduct better case studies with the industry.

Chapter 5

Graph-Based Verification of Program Constraints

To enhance reuse, one has to properly specify the constraints of the program. While reusing by extending or adapting an implementation of the program, it is important that the software engineers satisfy these constraints. Violations of these constraints may introduce errors to the program hampering the benefits of reuse. In addition to program constraints, the developers also has to follow the coding conventions enforced by their organization. Usually, a complex program has too many constraints and the coding conventions are used very frequently. So, it is a cumbersome task to manually verify all these constraints and coding conventions.

Obviously, in the literature, there is vast amount of research on formalizing requirements and verification of requirements with respect to the software system. From these, we particularly focus on program constraints verification. In this domain, analyzing the abstract syntax tree (AST) has pulled great attention [95, 36, 65, 39, 47]. In these systems the elements of the AST are converted to predicates and constraints are programmed using logic based languages. This is quite practical because the verification can automatically be derived from the software. Although these approaches are practical, they have the following drawbacks:

- The program elements in the AST are different from the source code. Constraints on comments and macros, for example, cannot be expressed or checked at the AST.
- In general, the programs that verify constraints tend to contain many predicates. Therefore, it is hard to comprehend the meaning of these programs and how they are applied on the ASTs.

- When a violation is located by these approaches, this violation is reported to the user with elements from the AST. Due to this, it may be too hard to locate the error in the source code. It is more beneficial to provide a description about the error and the location of the error based on the source code.

The contributions of this chapter are a process and a tool that use meta-modeling and graph pattern searching for providing automated detection of static program constraints and coding conventions violations. We use a two tier approach; in the first tier, the parts of the source code that violate the constraints are detected [33]. We utilize an extensible meta-model, called Source Code Modeling Language (SCML), for representing the source code as seen by the developers to address the expressivity problem. The models in SCML are attributed graphs, in which the source code elements are represented with nodes and the relationships between these elements are represented by edges. This, naturally, allows graphs to be used for visualizing the models in this meta-model.

Once the source code and the constraint violations are expressed in SCML, the problem reduces to searching for the occurrence of the constraint violation. The graph formalism has well established methods for searching for occurrences of patterns in the graphs. From these methods, graph transformations are used in the literature [100, 104] for searching for patterns in software artifacts and there are many widely used graph transformation tools. Because of these reasons and because models in SCML are attributed graphs, we use graph transformations to detect constraint violations. In our approach, the left hand-side of the transformation rule is used for detecting the violation of the design constraint and the right hand-side is used for collecting information about the location of the constraint and coding convention violation.

The graph representation of the source code may contain too many nodes/edges and, so, it may be hard for the stakeholders to spot the information about the program constraint and the coding convention violations collected by the graph transformation rules. As a consequence, there is need for a way to query the transformed graph and display the information collected by the transformation rules. Furthermore, some constraints can be a combination of other constraints. Predicate logic seems to be a good way from expressing the combination of the constraints. Because of this, in the second tier of our approach, we built the querying system with a the well-known predicate logic system Prolog [38]. Compared to other approaches, in our approach the predicate logic is only used to present the information extracted by the graph transformations in a convenient way. In our approach, when Prolog evaluation reaches certain predicates, the graph is searched for the nodes that are added by the transformation rules of the the first tier. The attributes of these nodes contain information on the constraint violation and these values are returned to the

Prolog system.

We developed dedicated parsers based on for generating the models from C++, C with preprocessor declaratives and Java. By extending these parsers and the meta-model of SCML, the process can be used with other languages. We use GROOVE [111] as the graph transformation tool. We extended GROOVE so that the model of the source code can be loaded without using intermediate tools. We also tied the Prolog engine to GROOVE so that it is possible to enter Prolog queries from GROOVE.

This chapter is organized as follows: the next section provides the example from an industrial software system that is the driving force behind the development of this approach. Section 5.2 describes the process in detail. Section 5.3 presents two case studies: one conducted on an open source software, the other one conducted on an industrial software software. The work that is related to our approach is described in section 5.4 and, lastly, section 5.5 provides the conclusions and the future-work.

5.1 Motivating Example

In this section, we show an example program constraint from a Magnetic Resonance Imaging (MRI) software system. This program constraint is frequently used in the control software of the amplifiers. The MRI machine contains amplifiers that are turned on/off at strict time intervals during a scan. The control software is responsible for reading/writing the values to the registers of the amplifiers at the right intervals. If the control software does not read a value at the right interval, the value may become outdated, causing errors in the scan process.

The amplifier keeps track of its status in registers called *general status registers*. The amplifier is real-time hardware; in that, when a request to read a register is made, the amplifier returns the value within a strict time limit. If the software reads the value after this limit, then the value may become outdated.

The communication with the amplifier is divided into parts called *sector*s. A sector consists of a request, a duration and a register set. Here, the request is a byte designating the operation of the amplifier; each operation has a unique number. The duration is the time interval after which the value of a register is outdated. The operations of the amplifiers take parameters by reading the values written to the status registers by the software; an operation reads a specific set of registers. Similarly, the operations write their return values to these registers. The register set in a sector is a map for defining the registers to which values are written (or from which values are read).

```
1: general_stati(DEVICE_STRUCT *device_ptr){
2: ...
3: GEN_SEOS(GENERAL_STATI, device_ptr, cs_ptr, GENERIC_DUR)
4: ...
5: }
6: analog_samples(DEVICE_STRUCT *device_ptr){
7: ...
8: cs_ptr->enableADC = true;
9: GEN_SEOS(SAMPLE_FIRST, device_ptr, cs_ptr, SAMPLE_FIRST_DUR)
10: ...
11: }
```

Figure 5.1: Two example uses of the sector generation macro; for each macro the constraint is to set the right duration for the right request.

Every different amplifier model is operated by a different control software. The control software is a bridge between the MRI software and the amplifier hardware. It has an interface of 32 functions and from these functions the MRI software controls the amplifier. These functions are fulfilled by at least one of the amplifier's operations. So, an interface function is implemented by initializing sectors that request the desired operations from the amplifier. In order to implement the control software, the developers have to implement the sector initialization code for with the right values in each of these interface functions.

Because the code for sector initialization is repeated a lot, the designers decided to implement it as a macro called *GEN_SEOS*. Figure 5.1 shows two uses of the macro *GEN_SEOS* (note that this code segment is a simplified version of the actual code segment). The first use, line 3, is to generate sectors for reading the general status registers. The request here is *GENERAL_STATI* and the duration for this request is stored in the macro *GENERIC_DUR*. The second use is at line 9, where the request is called *SAMPLE_FIRST* and the delay it takes to fulfil this request is stored in the macro *SAMPLE_FIRST_DUR*. Besides the duration constraint, line 8 shows another constraint of the request *SAMPLE_FIRST*: the amplifier's analog-to-digital (ADC) converter should be enabled in order to complete this request.

As can be seen from the two examples described above, setting up the registers and the delays for the requests are crucial constraints for the correct operation of the control software. The macro *GEN_SEOS* is used 16 times in the latest implementation of the control software; so, it is time consuming to check each usage for the constraints of the requests manually. The MRI software is a huge system and is tested rigourously with organizational policies like nightly build/tests. An overnight testing for the parameters of the macro is too much time consuming for a nightly

test. The developers need a way to check these constraints quickly without going through runtime tests.

In the remaining sections, we describe how such checks can be automated with computers on the source code. The example constraint on the initialization of the sectors *SAMPLE_FIRST* is used as a running example throughout the chapter. Note that the constraint on the request *SAMPLE_FIRST* requires the statement for enabling the ADC should come right before the reference to the macro *GEN_SEOS*. A variant of this constraint is that the ADC initialization and the reference to the macro *GEN_SEOS* to be on the same block of statements. With our approach, this variant can also be checked as described in Section 5.2.2.

5.2 The process for computer-aided Design Constraint checking

The process for computer-aided constraint verification (CACV) is a two-tier approach: in the first tier the program constraint and coding convention violations are detected and the error report is prepared. The second tier provides means for searching for a combination of the program constraints and/or coding conventions violations. In the rest of this chapter we refer to program constraints and coding conventions only as constraints.

In the process, constraints are modeled as graph transformation rules with a modeling language called Source Code Modeling Language (SCML) that includes the program elements at the source code. Graph transformations provide a well-defined formalism to search for patterns and there are many mature tools that support graph transformations. A graph transformation rule r has a left hand-side, L, and a right hand-side, R. In order to transform a host graph, with rule r, L should exist in the host graph (interested readers are referred to the literature for the formal definition [46]). In our approach, the left-hand side of the transformation rule models the constraint. If the left-hand side exists in the graph model of the source code then the constraint is violated. The right-hand side adds graph elements (with attributes) that describe the constraint violation, the location of the violation and the other information (like the names of the program elements) that is needed to be presented to the developer.

The process makes use of a repository to manage/store the modeled constraints and conventions; we use the Computer-Aided Design Evolver tool (CDE) [32] to manage the repository. In the first tier, the developer checks-out the transformation rules modeling the constraints from the repository. The transformation rules in the

repository are stored as templates; that is, the names of the program elements are parameterized. If the transformation rule the developer wants to check-out contains parameters, then CDE asks the real values of these parameters from the developer. With the supplied names CDE binds the constraint models.

The source code that implements the design is converted to an SCML model, which is an attributed graph. We programmed dedicated converters for Java and C (with preprocessor directives) for this purpose. We also implemented a proof-of-concept (i.e. it cannot parse all statements) converter for Java using the generic text-to-model textual concrete syntax (TCS) [75]. Once the source code is converted and the transformation rules are bound, the constraint violations can be detected with a graph transformation tool; we use GROOVE [111].

If a constraint is violated, GROOVE automatically applies the transformation rule which models that constraint. The transformation rules in the process add nodes to the graph that contain a description of the constraint. In addition to the description, the SCML preserves the physical line numbers of the statements; so, the rules copy the line numbers of the statements violating a constraint to the node they add. Thus, the developer can learn about the constraint and the line numbers of the statements violating the constraint by locating the node added by the transformation rules.

We added a second tier to CACV that provides querying for constraints, for the following reasons:

- A constraint can be a combination of other constraints. For example, both $pattern_1$ and $pattern_2$ should be in the design.
- The developers can also query for parts that obey the constraint and, when they do, they should get an output stating the constraint is not violated.
- The patterns observed from the source code may be too low level and we need a way to express the constraint at a higher level and convert them to the patterns of the design.

We use the well-known logical programming language Prolog to provide the querying. In the querying system we developed, there are predicates representing the nodes and edges of the graphs. When the evaluation of a Prolog rule reaches these predicates, they execute code in GROOVE that traverses the graph to find the description nodes/edges added by the transformation rules.

In addition to providing a storage for the modeled constraints, the repository also hides the underlying graph transformation and Prolog files from the user. In the repository the constraints are stored and retrieved with descriptions. For example,

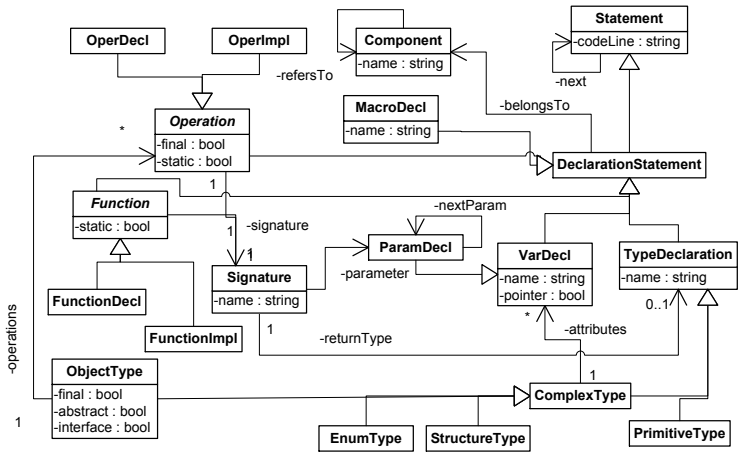


Figure 5.2: The meta-model of the declaration statements in the Source Code Modeling Language(SCML).

all the constraints that a component of a software should follow can be given the name of the component. The constraints that are a combination of other constraints are expressed as Prolog rules and stored in the repository as Prolog source files. We extended CDE to parse the Prolog source file and retrieve the transformation rules the Prolog file is referring to from the repository. Thus, whether the constraint is a transformation rule or a Prolog file is transparent to the user; the user only supplies the names of the constraints to CDE.

In the rest of this section, we describe how the constraints are modeled as graph transformations rules and how Prolog is tied to the transformation rules.

5.2.1 Source Code Modeling Language

This section describes the meta-model of SCML; the models in this language are attributed graphs. SCML can be used to represent source code in C/C++ with preprocessor declaratives and Java as graphs. Note that we do not provide the full SCML meta-model in this section, we only describe the elements used in the figures in this and in the next chapter.

In the SCML meta-model the abbreviation *Decl* stands for declaration, *Impl* stands for implementation, *Var* stands for variable and *Oper* stands for operation (i.e. method). The nodes labeled as *Component* refer to source files. The components can refer to other components; this is used for modeling *includes* and *usepackage* relations between source files. Every software entity that belongs to a component is a *Statement*. The ordering between the statements is modeled with the edges labeled

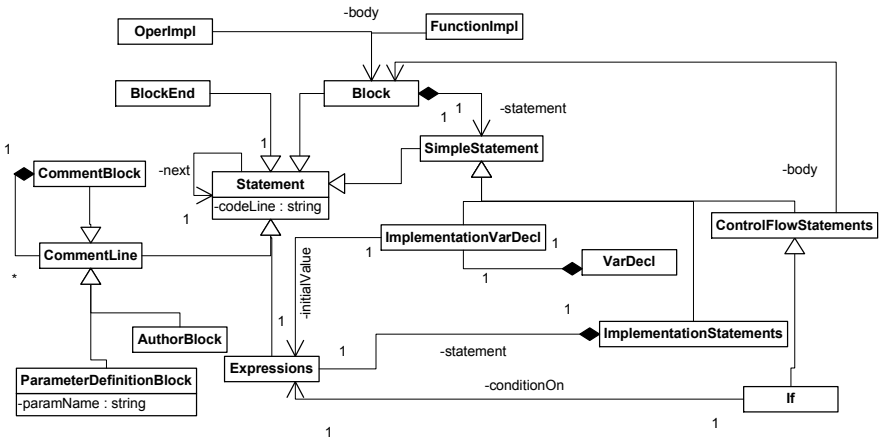


Figure 5.3: The meta-model of expressions, comment lines and the statements for modeling function/method bodies in SCML

next. There are four kinds of statements: *declaration statements*, *comment lines*, *simple statements* and *expressions*. The declaration statements are statements used for declaring software entities; these statements are not used within a function or method. Figure 5.2 presents the meta-model for the declaration statements. A software entity is declared in a source file; this is modeled by connecting the respective declaration statement to the component node with an edge labeled *belongsTo*.

The meta-model supports variable declarations (nodes with label *VarDecl*), type declarations, macro declarations (nodes with label *MacroDecl*) and function/operation declaration statements (nodes with label *FunctionDecl* and *OperDecl*). All types have designated names modeled as an attribute of the type node. The nodes labeled as *PrimitiveType* refer to data types like *int*. The model supports three complex data types: enumerations (nodes labeled *EnumType*), structures (nodes labeled *StructureType*) and object type nodes (nodes labeled *ObjectType*). All these types have attributes that are variable declaration nodes. Only object types and structures (depending on the language) can be connected to method declarations or method implementations with edges labeled *operations*, which shows that the object type either declares or implements the method.

SCML differentiates between function/method declarations and implementations because: 1) the component that declares the function can be different from the component that implements it; 2) the object type that declares the method (abstract methods) can be different from the object types that implement the method. Nodes labeled as *Signature* represent method/function signatures; the attribute *name* designates the name of the signature. The parameters are modeled with edges connecting the signature node to parameter declaration nodes (nodes with label *ParamDecl*)

labeled *parameter* and the return type is modeled with an edge connecting the signature node to a type declaration node labeled *returnType*. The order of the parameters is modeled with the edge labeled *nextParam*. The method/function declaration or implementation nodes are connected to signature nodes with edges labeled *signature*.

Figure 5.3 represents the meta-model of the other kinds of statements. The body of a function/method implementation is represented by a block statement that is connected to respective function/method implementation by an edge labeled *body*. The statements of a function/method are represented by nodes labeled *simple statement*. There are three kinds of simple statements: the first kind represents terminated expressions (i.e. terminated with *;*) and this kind is called implementation statements (nodes labeled *ImplementationStatements*), the second kind represents the control flow statements (nodes labeled *ControlFlowStatements* such as an *if* statement) and the third kind represents the variable declarations within functions/methods (nodes labeled *ImplementationVarDecl*). The control flow statements are followed by block statements (i.e. they have bodies). An implementation statement has an expression, which it terminates. So, in SCML models an implementation statement node is also labeled with the expression it is terminating. For example, a function call expression that is terminated (e.g. *foo();*) is represented by a node that is labeled *ImplementationStatements* and *FunctionCall*. Every statement of a block is connected to the respective block node with an edge labeled *statement*. The last statement of a block is connected to a block end statement with an edge labeled *next*.

Figure 5.4 presents the meta-model of expression statements, which includes the following expressions: function call (modeled by nodes labeled *Call*), method call (modeled by nodes labeled *MethodCall* and for static methods *StaticMethodCall*), object create (modeled by nodes labeled *CreateOper*), macro reference (modeled by nodes labeled *MacroReference*), return (modeled by nodes labeled *return*) assignment (modeled by nodes labeled *Assignment*) and value (modeled by nodes labeled *Value*).

The parameter element is a special expression that is composed of other expressions and can be connected to another parameter element with the edge labeled *nextParam*. The edge *paramValue* from a call node to a parameter node is used for modeling the parameters passed by a call expression. The order of the parameters is modeled by an edge labeled *nextParam*. The first parameter of the call does not have an incoming edge labeled *nextParam* and the last parameter of the call does not have this edge as an outgoing edge.

The nodes labeled *ValueOf* and *AddressOf* are used to represent a reference to a variable or the value of another expression (like function call). These nodes can either be connected to other expression nodes or variable declaration nodes with an edge labeled *referenceVar*. A *ValueOf* connected to a variable declaration node

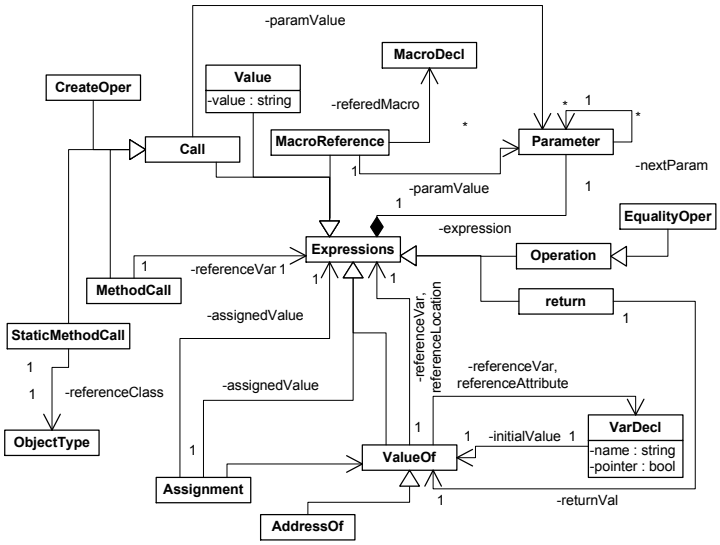


Figure 5.4: The meta-model of the specializations of the expression in SCML.

represents that the variable is referred to. Similarly, a *ValueOf* node connected to an expression node means that the resulting value of the expression is referred to (i.e. the return value of a function/method call). These nodes can be connected to attributes of complex types (like a structure) with an edge labeled *referenceAttribute* to model that the value (or the address) of the connected attribute is referenced. The reference to the value (or the address) of an array cell is represented in SCML by edges labeled *referenceLocation* connecting a *ValueOf* (or an *AddressOf*) node to an expression or a variable declaration node. A variable declaration statement can be connected to a *ValueOf* expression with an edge labeled *initialValue*. This this means that the initial value of the variable is set to the expression whose value if referred.

SCML treats the parameter of calls or macro references also as expressions. Thus, for example, the parameter of a call statement can also be a call statement. And since *ValueOf* and *AddressOf* are also expressions, a parameter can also be a reference to a variable.

The left and right hand-side of an assignment expression is always a *ValueOf* expression (or one of its subclasses); this means that the value of the referred expression is used by the assignment statement. With this, the return value assignment of a function call can be modeled, for example, by connecting the assignment statement node to a *ValueOf* node with an edge labeled as *assignedVar* and connecting this *ValueOf* node to a call expression.

The constant values are represented with by labeled *Value* in SCML. The attribute

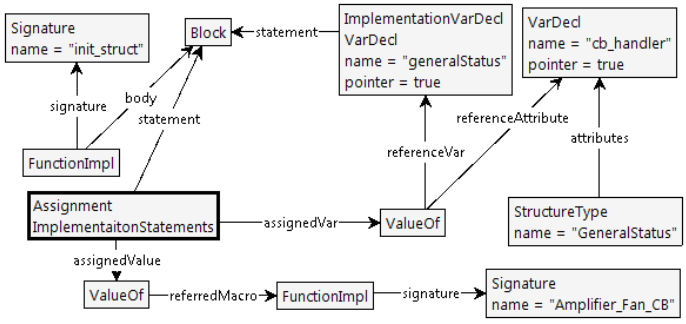


Figure 5.5: The attributed graph in SCML representing the assignment statement *generalStatus->cb_handler = Amplifier_Fan_CB*

value is the constant value this node is representing. The constant values are treated as expressions, so other expressions can refer to them. For example, an initial value of a variable can be set to a constant.

To better explain the *ValueOf* statements, we show how the program fragment, containing an assignment statement, given below is modeled using SCML:

```
init_struct(){
...
generalStatus->cb_handler = Amplifier_Fan_CB
...
}
```

Figure 5.5 presents the attribute graph in SCML modeling this statement. Here, the assignment statement is the emphasized node. The left hand-side of an assignment is designated with the edge labeled *assignedVar*. In the figure, left hand-side of the assignment statement is connected to a node labeled *ValueOf*. The ValueOf expression in the figure is connected to the variable *generalStatus* with an edge labeled *referenceVar* and to the attribute *cd_handler* with an edge labeled *referenceAttribute*. This means that the instance of the structure *GeneralStatus* the variable *generalStatus* holds is accessed and from this instance the value of the attribute *cd_handler* is referred. Since this ValueOf expression is at the left-hand side of the assignment, the value at this attribute gets assigned.

The right hand-side of an assignment statement is represented by an edge labeled *assignedValue*. In the assignment statement from the example program fragment, the right-hand is again a ValueOf expression. This time, however, the the value referred is the value of the function implementation *Amplifier_Fan_CB*. Thus, the assignment statement assigns the value of this function implementation (which is an address) to the attribute *cb_handler*.

SCML is designed to cover elements from both procedural (e.g. C) and object-oriented languages (e.g. Java). We made such a design choice because, usually, industrial software is written in more than one language and programming paradigm. Although, we do not present examples of program constraints on statements from multiple languages, one can generate an SCML model from C and Java source code and express a constraint that checks the statements of these two languages at the same time.

Source Code to Graph Conversion

We programmed dedicated source-to-SCML converters for Java, C/C++ (that can detect macros). Here, the Java parser is based on ANTLR [1]. The C/C++ parser, on the other hand. is written in C# where a C/C++ ANTLR [1] grammar is used as a reference. The SCML treats every terminated statement (or implementation and declaration statement) as an entity because we wanted to apply SCML to source code where statements from other languages can exist (e.g. a domain specific language) or generate models where only a certain set of statements are parsed (e.g. call and assignments are parsed). Such a model requires a parser that runs in two levels with two sets of grammars: the first set describes what a terminated statement is and the second set describes how each terminated statement is parsed. Because the Java application had only minor changes to standard Java grammar, we were apply to use a parser based on Antlr. However, the industrial case studies used in Chapter 6 have statements from a domain specific language which needs to represented in the SCML model to preserve the completeness of the source code, we programmed a dedicate parser based on a reference grammar. We defined the terminated statements as references to certain macros, comment blocks, comment lines, the reserved words (such as if and while), block statements and statements that end with ";". Then, we implemented methods for parsing the terminated lines (i.e. C expressions); if the parser cannot parse a terminated statement, it converts this statement to an abstract implementation statement node whose attribute *codeLine* is set to the statement itself. Because of the dependencies on the header files requires the include path resolution, the parser simply asks the user to specify the declarations of the type names if it cannot find the include file declaring the type in the same directory as the source file. Figure 5.6 shows this procedure where the parser is asking the user to specify the declaration of the type name *MGOBJGI_AMP_MODE_ENUM*.

In the literature, there are also general purpose source-to-model converters [75, 71]. These converters take the meta-model and a grammar as inputs and they automatically generate the source-to-model converter tools. The major benefit of these tools is that extensions to the meta-model require only the extensions to the grammar and

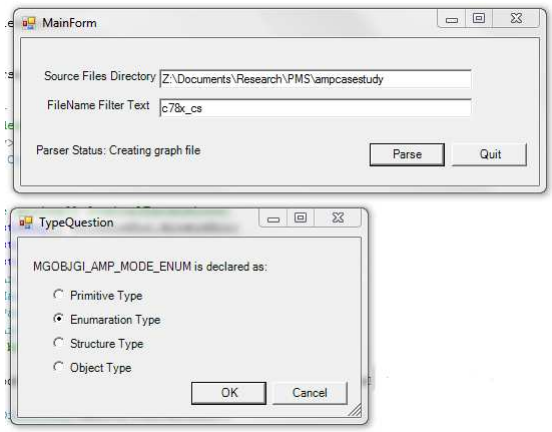


Figure 5.6: The dedicated C/C++ parser resolves the declarations for types by asking the user to specify the declaration.

the meta-model specification (i.e. no conversion algorithm is implemented). Thus, one can implement a SCML converter for a language by just specifying the grammar of the language; the parser and the model converter would be generated automatically. For our approach, we implemented the SCML meta-model in KM3 [74] and implemented a proof-of-concept Java grammar in TCS (this proof-of-concept implementation can parse class declarations, attribute declarations, method declarations, assignment statements and method call statements). We extended GROOVE so that the generated models (XMI files) can be loaded from GROOVE.

The TCS language does not support scoping; for example, if two functions declare a variable with the same name then TCS cannot differentiate these two variable (and gives a parse error). We addressed this problem by resolving the variables and called signatures when the model is loaded to GROOVE. The source-to-model converter generated from TCS, for example, does not resolve to which variable a statement refers, it only records the name of the variable that is referred. When the model is loaded into GROOVE, the loader traverses upwards from the statement with the variable reference to the *component* node to identify where the variable is declared.

Preserving the Source Code Locations

When a constraint violation is detected, it is important to provide guidelines about the possible location of the problem to the user. We use the physical line numbers for providing such a guideline. The integer attribute called *lineNumber* is added to the simple and declaration statement nodes for storing the line number is in SCML models. The line numbers are extracted from the source code during the

source-to-model conversion.

5.2.2 Modeling Constraints with Graph Transformation Rules

The template for modeling a transformation rule that detects a constraint violation is as follows: the expressions/statements that are needed to identify the location (or a usage) of the expressions/statements on which a constraint is defined are modeled in SCML and placed in the left hand-side of the transformation rule. The expressions/statements that follow the constraint are modeled in SCML and placed in the rule as negative application conditions [64] (when the whole pattern depicted by these nodes/edges occurs in the host graph the rule does not match). The right hand-side of the rule adds a node labeled *constraint* (we refer to these edges as constraint nodes), this node is connected to the simple or the declaration statement(s) where the constraint is violated by an edge. The label of this edge is the name of the constraint and it is used in querying for the constraint in the second tier. The text describing the constraint is added as the attribute *description* to the constraint node. Here, the rule can be programmed to concatenate this text with the names of the program elements (an example of this is shown in section 5.3). Lastly, the rule is modeled to copy the attribute line-number from the implementation and/or the declaration statement(s) where the constraint is violated to the attribute *problemLine* of the constraint node.

In Section 5.1, the two constraints of the sector initialization for the request *SAMPLE_FIRST* were shown. The first constraint is that duration of the constraint is *SAMPLE_FIRST_DUR* and the second constraint is that the amplifier's ADC is enabled (line 8 of Figure 5.1). Figure 5.7 presents a graph transformation rule in SCML, modeled according to the template described above, used for detecting the code segments that generate the sectors for this request without satisfying these constraints.

To remind the reader, in GROOVE both left hand-side and right hand-side of the graph transformation rule are shown in the same graph, using some notational constructions. The dashed (red) nodes and edges present the negative application conditions [64]. Thick (green) nodes and edges are graph-elements that are added by the transformation rule. All nodes and edges except the circular and the thick (green) ones belong to the left hand-side of the rule. The right hand-side contains all the edges and nodes that are not dashed; the meaning of the circular nodes are described later in this section. GROOVE is a production simulator; for a graph transformation system it generates a state-space whose transitions are labeled by the names of the transformation rules that matched.

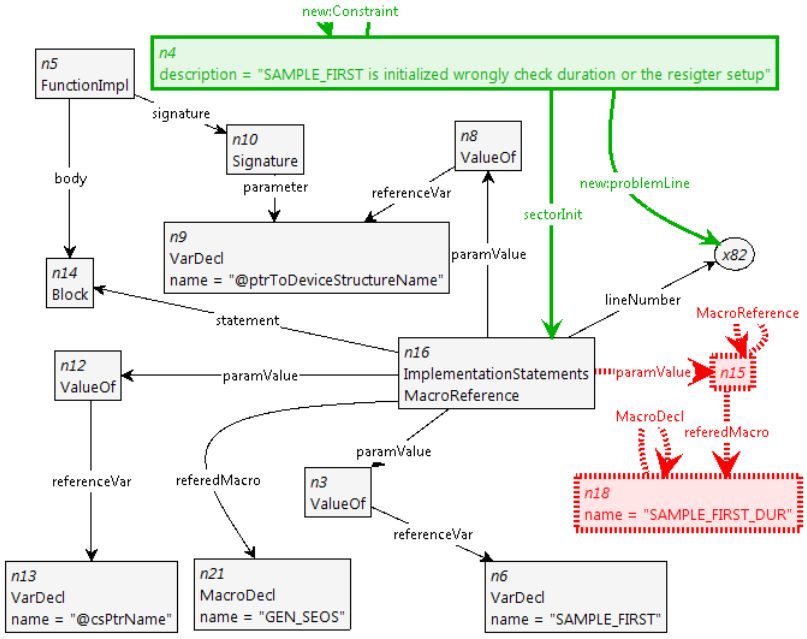


Figure 5.7: The graph transformation rule used for checking the constraints for the request *SAMPLE_FIRST*

In Figure 5.7, the node $n16$ is a reference to the macro *GEN_SEOS*. The first parameter of this macro is the value of the variable *SAMPLE_FIRST* (node $n6$). This means that the sector generation is used for the request *SAMPLE_FIRST*. The nodes are in the left hand-side of the rule, because they are required to identify/locate the usage of the macro *GEN_SEOS* with the parameter *SAMPLE_FIRST*. The last parameter of the macro *GEN_SEOS* is a macro reference expression (node $n15$) referring to the macro *SAMPLE_FIRST_DUR* (node $n18$). These nodes are negative application conditions of the transformation rule and with these nodes and edges the rule detects the violation of the duration constraint of the request. If the last parameter is a macro reference expression to the macro *SAMPLE_FIRST_DUR* then the rule does not match. If, on the other hand, the last parameter is another expression (e.g. a reference to a different macro) then the rule matches. This is because the request *SAMPLE_FIRST* is used without the duration *SAMPLE_FIRST_DUR* and, thus, the duration constraint of the request is violated.

The right-hand side of all transformation rules used for detecting the violation of a constraint adds a node labeled *Constraint* (constraint node) connected to the statement node that violates the constraint. The edge connecting the constraint node to a statement node is labeled with the name of the constraint. A constraint node has two attributes: an integer attribute named *problemLine* and string attribute labeled

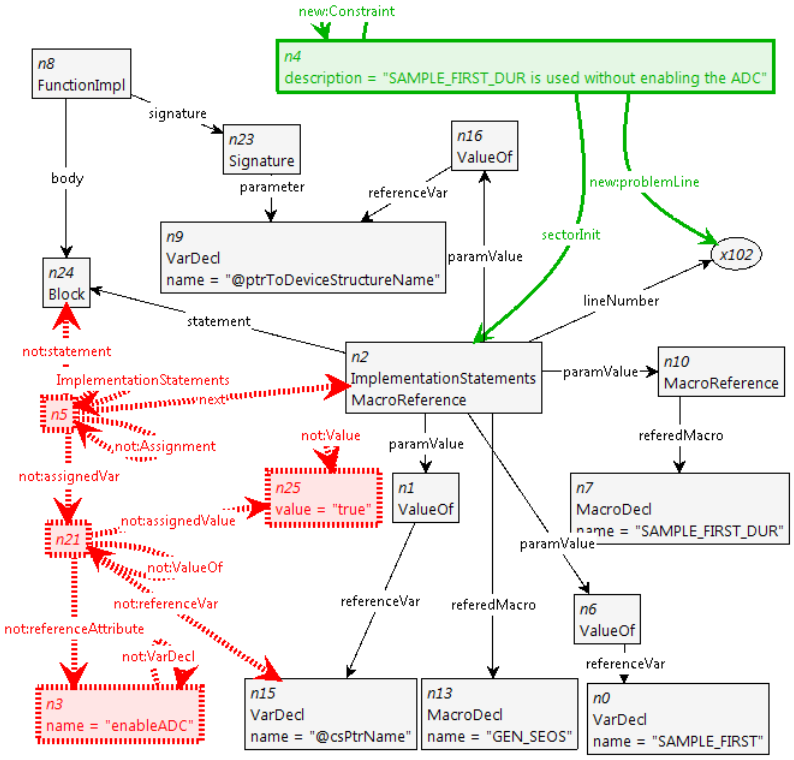


Figure 5.8: The graph transformation rule that checks if ADC is enables when the request *SAMPLE_FIRST* is used

Description. The value of the attribute *Description* is set to a descriptive text about the constraint when the rule is modeled. The attribute *problemLine* is set to the line number (the attribute *lineNumber* of a statement node) when the rule is applied. In Figure 5.7, the constraint node that is added when the rule is applied, is the node $n4$ and it is connected to the macro reference (node $n16$) (line 9 of Figure 5.1). The value of the attribute *Description* is hard coded in the rule; however, using attribute operations this string can be, for example, concatenated with names of the program elements (an example of this is presented in section 5.3.1). The edge from the macro reference node $n4$ labeled *lineNumber* represents the attribute with the same name. This edge is connected to the node $x82$, which represents the value of the attribute *lineNumber* and it matches to any value of the attribute. The rule adds an edge labeled *problemLine* from the constraint node to the value node of the attribute *lineNumber*. This sets the value of the attribute *problemLine* to the line number of the macro reference. In this way, the error reporting also includes the line number where the violation has occurred.

Another constraint for the request *SAMPLE_FIRST* is the enabling of the ADC; in

line 8 of the Figure 5.1 this is done by assigning the value *true* to the field *enableADC* of the structure used for register mapping. If the request *SAMPLE_FIRST* is used without this assignment statement then the constraint is violated. The transformation rule checking whether the assignment statement comes right before the macro reference to the macro *GEN_SEOS* with the request *SAMPLE_FIRST* is presented in Figure 5.8. Note that here, the macro reference is the node $n2$ and the parameter to the value of the macro *SAMPLE_FIRST* is the node $n7$. Because the violation of the constraint happens when the request *SAMPLE_FIRST* is used without enabling the ADC, these macro references are placed in the left-hand side of the rule. The node $n5$ represents the assignment statement use for enabling the ADC; the right-hand side of the assignment is the value *true* (node $n25$) and the left-hand side of the assignment is the reference to the field *@csPtrName->enableADC*. These nodes, except the node representing the variable holding the register setup (named *@csPtrName*), are negative application conditions. This ensures that the rule only matches when the statement *@csPtrName->enableADC* is not the source code. Note that the rule does not check in which function/method these statements belong. It only ensures that the assignment and the macro reference belong to the same block (i.e. the same scope) and that the assignment statement comes right before the macro reference statement. The edge labeled *next* connecting the assignment statement node $n5$ to the macro reference node $n2$ is also a negative application condition; thus, if the assignment statement is not before the macro reference to *GEN_SEOS* then the rule marches. As discussed before a more relaxed version of this constraint is checking whether the assignment statement occurs in the same block but not directly before the macro reference with the request *SAMPLE_FIRST*. This constraint can be expressed by removing the edge labeled *next* between the nodes $n5$ and $n2$.

Graphs transformations with negative application conditions are expressive enough for searching for violations of program constraints with existential quantifier ($\exists$). That is, when one expresses for the existence of an violation of an statement for certain set of statements as shown in this section. However, a single graph transformation with negative application conditions is not enough for expressing constraints with the universal quantifier ($\forall$). Such constraints can be expressed using a stack of transformation rules as shown by Rensink et al. [109]. In CACV, we only focus on constraints with existential quantifiers. Nevertheless, the same tools and process can still be used when a constraint is expressed with a stack of transformation rules.

Parameterizations of Names

For some constraints, the names of the program elements they work on (e.g. the names of the variables or functions) or the values (i.e. the value attribute of a value

node) may change. Rather then modeling a different constraint for each possible name or value, we parameterize these.

A name or a value that is parameterized starts with the @ character. During the fetch of the transformation rules from the repository, the CDE tool forms a binding file containing all the parameters. The binding file contains a listing of the form *@parameter = givenName*. The developer fills this file. Then, the CDE tool binds the transformation rules by replacing the parameters with the values supplied by the developer.

In the example presented in Figure 5.1 the values stored in the variables *device_ptr* and *cs_ptr* are "passed" to the macros; however, the developers are free to change the names of these variables in the upcoming versions of the control software. If the names of these variables were fixed in the transformation rule, then the rule could not be used when the names of the variables were changed even though the same constraint still applies. In Figure 5.7 and Figure 5.8, the name of the variable that holds the register mapping (node $n13$ in Figure 5.7 and node $n2$ in Figure 5.8) is set to *@csPtrName* meaning that the name of this variable is parameterized. When the developer wants to check for this constraint, the binding file formed by the CDE tool contains an entry for this parameter and the developer has to supply the name used for this structure (e.g. *cs_ptr*).

5.2.3 Querying for Constraints: Connection of the Graph System with Prolog

After providing the actual names for the parameters of the transformation rules, CDE forms a graph production system containing the transformation rules the developer wants to verify and launches GROOVE. In GROOVE, the developer can load the model of the source code.

In order to find which constraints are violated, the graph production system is *simulated*. The simulation automatically applies the matching transformation rules. It also generates a state-space, in which the loaded source code is the start state and the transitions are the applied transformation rules. At certain states, no more transformation rules can be applied; these states are called *final states*. The final states are the graphs that contain all the constraint nodes added by the transformation rules.

Manually searching for the constraint nodes in a graph can be a cumbersome task, depending on the size of the source code. Obviously, there is need for a querying mechanism that automatically searches for the constraint nodes for the constraints

the developer is interested in. If such a constraint node exists then the line number and the description about the constraint is returned. If no such constraint node can be supplied then the constraint is not violated. We use Prolog to provide such a querying mechanism. We use Gnu Prolog [8], a Prolog engine written in Java. The major benefit of this engine is that the predicates can refer to Java methods. We tied this Prolog engine to GROOVE and extended GROOVE with panels from which the developer can enter the queries (or write Prolog programs).

The developer can search for a constraint violation with the predicate *constraint (Name, Line, Description)*. Here, the developer only needs to provide the name of the constraint as *constraint (Name)*. *Line* and *Description* are unbounded variables; their values are supplied at the end of the evaluation (the tool automatically convert a user-entered query to the Prolog query with the unbounded variables). Below the implementation of the predicate *constraint* is given:

```
constraint(Name, Description,Line):- graph(G),
label_edge(G,Name,E),edge_source(E,N),
node_self_edges(G,N,['Constraint']),
node_with_attribute(G,N,'Description',Description),
node_with_attribute(G,N,'problemLine',Line).
```

Here, the first predicate returns true when the graph G is a graph of a final state. The predicate *label_edge* looks for an edge, E, in the graph whose label is equal to the supplied value for the parameter *Name* and the predicate *edge_source* is true when the node N is the source of the edge E. The last three predicates are used for the properties of the nodes: the predicate *node_self_edge(G,N,L)* returns true when the node N in the graph G that has the list of the labels L and the predicate *node_with_attribute(G,N,A,V)* returns true when the node N in graph G has an attribute labeled A whose value is set to V. These predicates all refer to Java methods that search for the final states and the graphs for the provided methods. Moreover, these are the only predicates that are stored in the Prolog database; thus, we implement querying for a constraint in 6 predicates.

As discussed above, to search for a constraint the developer only enters the name of the constraint. The Prolog evaluation searches for the node labeled *Constraint* that is connected to a node with an edge that has the same label as the name the user entered in the graph of each final state. When such nodes exist in the final graphs, Prolog gets the values for the attributes *Description* and *problemLine*. Assume that the constraint on the request *SAMPLE_FIRST* described in Section 5.1 has been violated at line 142 and the transformation rule for this constraint is called *seos_sample* (Figure 5.7) and is applied. To query for the violations of this constraint the developer enters *constraint('seos_sample')*. Then, Prolog returns:

Line = 142
Description = SAMPLE_FIRST is initialized wrongly check duration or the resigter setup

This shows the developer that the assignment statement at line 142 is used without correctly initializing the structure general status.

5.2.4 Expressing Combinations of Constraints

Names of some constraints can be too low level and for certain stakeholders a transition from a high level constraint to a low level constraint may be needed. For example, the name of the constraint periodicCheckInitialization may be too detailed. It requires one to know the name of the edge added by the transformation rule. However, using Prolog rules it is possible to implement a transition mechanism that hides the name of the edge. Below is a Prolog rule that hides the name of the constraint:

```
wrongPeriodicChecking(Description,Line) :-
constraint('periodicCheckInitialization',Description,Line)
```

Some constraints can be a combination of other constraints. Rather than modeling a new graph transformation rule for these constraints, they can be expressed in terms of other constraints also by using Prolog rules. In section 5.3.1, we present examples of such constraints.

5.3 Application of the Approach

We applied the CACV approach to two different software systems: 1) an open source ECORE model transformation tool 2) the gradient amplifier control software.

For the first software system, we verified that several program constraints are not violated. This ECORE model transformation tool has to evolve every time the meta-model changes and there are strict constraints the developers have to obey for successful operation of the software. In the next subsection, we describe how we modeled an important constraint of the ECORE converter tool.

A crucial aspect of our approach is evolution: even though the software evolves, the constraints do not change and the developer has to respect them for correct evolution of the software. Thus, it is important to see how many of these constraints stay valid through evolution. With the gradient amplifier software, we modeled the constraints

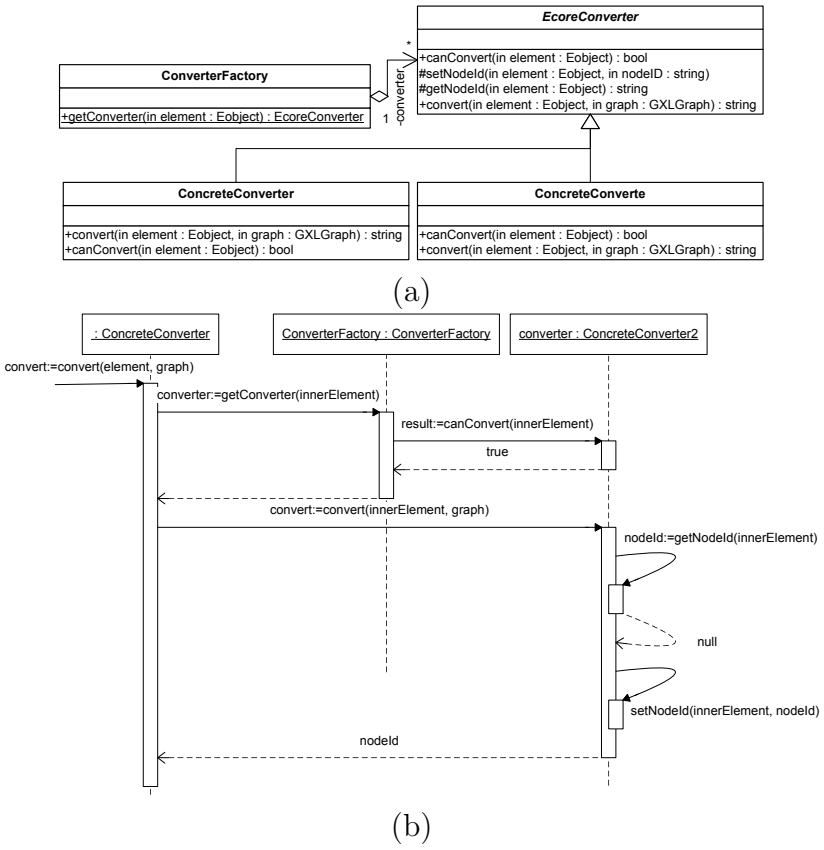


Figure 5.9: (a) Class Diagram of the converter tool with two converters. (b) The sequence diagrams showing an conversion scenario with two converters.

for the latest amplifier. Then, we verified that these constraints are not violated for the control software of the first and second amplifier models. Subsection 5.3.2 presents our findings.

5.3.1 ECORE to GXL model transformer tool

The ECORE to GXL converter is an open-source tool used in various research studies and experiments (the tool can be downloaded from [7]). The converter is written in Java and consists of 32 classes and 1586 lines of code. With the developers of the tool, we identified 8 constraints which are captured with 11 graph transformation rules. Here, we describe three of these constraints that are very crucial for the correct operation of the converter.

Figure 5.9-(a) depicts a stripped version of the class diagram of the converter tool.

For each element in the ECORE meta-model there is a respective converter class that deals with converting the ECORE object to its graph equivalent. However, a converter class can be responsible for converting more than one ECORE class. Each converter class specializes the abstract class *EcoreConverter*. This abstract class implements the methods *setNodeId* and *getNodeId*. In the graph model a node is identified by a unique identifier and the ECORE objects (defined in the meta-model) have an attribute called *id* that holds this unique identifier. The identifier is set by the converter tool during conversion and is used to prevent converting the same ECORE object to graph node more than once.

The abstract class *EcoreConverter* has two abstract methods: the methods *canConvert* and *convert*. In the method *convert* the conversion algorithm is implemented. The method *canConvert* is used to identify which ECORE object the class converts. Figure 5.9-(b) depicts an example conversion scenario, where an instance of the class *ConcreteConverter* is converting an ECORE object. The object has connections to other ECORE objects and to convert these connections, concrete converter asks the class *ConverterFactory* to return the converter for this object. The *ConverterFactory* iterates through the concrete converter class instances and by calling the method *canConvert* it identifies the converter that can convert the ECORE object. Here, this method checks if the name of the ECORE object's class is equal to the name set in the converter class. Once the converter is identified, the *convert* method of the converter is called. In each converter, the convert method has to call the method *getNodeId* to see if the ECORE object is already converted. If not, then during the conversion the method *setNodeId* should be called to set the identifier of the ECORE object.

The presence of the calls to the methods *setNodeId*, *getNodeId* and the presence of the comparison of the names in the *canConvert* method are constraints that should be obeyed by every converter. We modeled 3 transformation rules for these constraints:

I *checkSetNodeId*: this constraint checks if the method *setNodeId* is called from the method *convert*. The transformation rule matches only if the call to the method *setNodeId* is not present in the method *convert*. The parameter of this rule is the name of the ECORE class to be converted.

II *checkGetNodeId*: this constraint checks if the method *getNodeId* is called from the method *convert*. The transformation rule matches only if a call to the method *getNodeId* is not present in the method *convert*. This rule has one parameter and it is the name of the ECORE class.

III *converterEClass*: This constraint checks if the converter class correctly identifies the ECORE object it can convert. The identification is done by checking

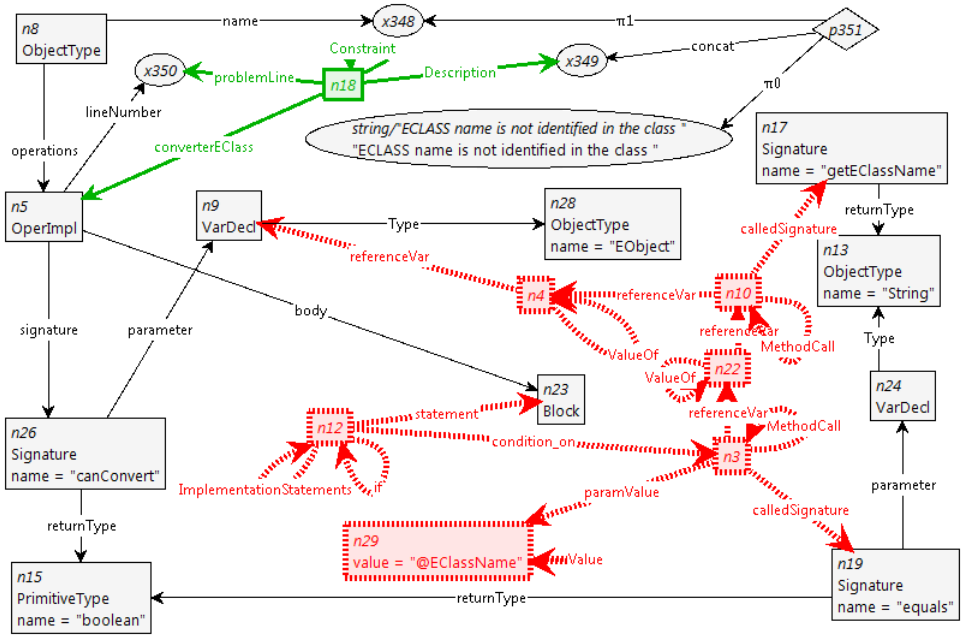


Figure 5.10: The graph transformation rule that checks if the converter class specifies the *EClass* name correctly.

if the ECORE object's (EObject) EClass name is equal to the name specified in the converter class (this value is specified at compile-time through the call *EObject.geteEClassName().equals(EClassName)*). The transformation rule of this constraint is depicted in Figure 5.10. This rule matches when the calls to identify the name of the EClass are not made, or when the specified EClass name is correct or when the identified name of the ECLass is not compared with the specified name.

In this transformation rule, the node *n24* represents the *if* statement belonging to the method *canConvert*. The condition of the *if* statement is a method call expression; the method call node *n3* in the figure. This method call refers to a return value of another method call expression, which the node *n10*. The second call statement (node *n10*) is a call to the method *EObject.getEClassName()* and the first call statement (node *n3*) is a call to the method *String.equals()*. The parameter to the second call statement is a value (node *n20*) that is equal to *@EClassName*. This is a parameter for the name specified in the converter class at compile-time.

The right hand-side of the rule adds the constraint node *n*6. The constraint node is connected to the operation implementation node representing the method *canConvert* and the attribute *problemLine* is set to the line number of this node.

In this way, the rule shows that the method *canConvert* violates the constraint *converterEClass*. The class where this constraint is violated is also an important information; because the name of the class where the violation occurs can change, the transformation rule does not fix a description text. The value of the description text is computed by the rule by concatenating the text *ECLASS name not identified in the class:* with the attribute node $x303$ which contains the name of the converter class (where the violation has occurred). The concatenated string is then stored at node *x306* which is the attribute that stores the problem description of the constraint node.

Using the system it is possible to query for each of these constraints. However, a stakeholder may only be interested in knowing whether a converter class obeys all constraints or not. The following Prolog rule is used for this:

```
converterClass(ErrorDescription,LineNumber) :-
constraint('checkSetNodeId',ErrorDescription,LineNumber);
constraint('checkGetNodeId',ErrorDescription,LineNumber);
constraint('converterECLass',ErrorDescription,LineNumber).
```

This rule evaluates to true if a converter class violates to at least one of these constraints. In this case, the description and the line number for the constraint violation is printed. If none of these constraints are violated Prolog just returns *false*.

The transformation rules modeling the constraints and the Prolog rules are placed into the repository using the CDE tool. To check for design constraint violations, the stakeholder (e.g. the developer) runs CDE with the name of the constraint(s). For the ECORE to GXL converter, for example, the stakeholder runs *cde -constraint converterClass*. The CDE tool, then, checks if Prolog rules or graph transformation rules matching the name of *constraint* argument exist in the repository. If they exist, the *CDE* tool asks the stakeholder to supply the parameters (if the transformation rules have parameters). Because the constraint *converterFunctional* is a Prolog rule, the CDE tool looks for all transformation rules this rule refers to. If all rules are present in the repository, then a list of parameters is formed and presented to the user. In this case, the user is asked for the parameter *@EClassName* (note that it is currently not possible to use the same constraint more than once due to parametrization). Once all parameters are supplied, the CDE tool launches GROOVE.

Figure 5.11 presents a screenshot of GROOVE with the *3* constraints explained in this section loaded. Here, the user queried for all the converter classes that violate at least one of the three constraints described. None of the transformation rules has matched/applied to the GSML model of the converter class in question. As a result,

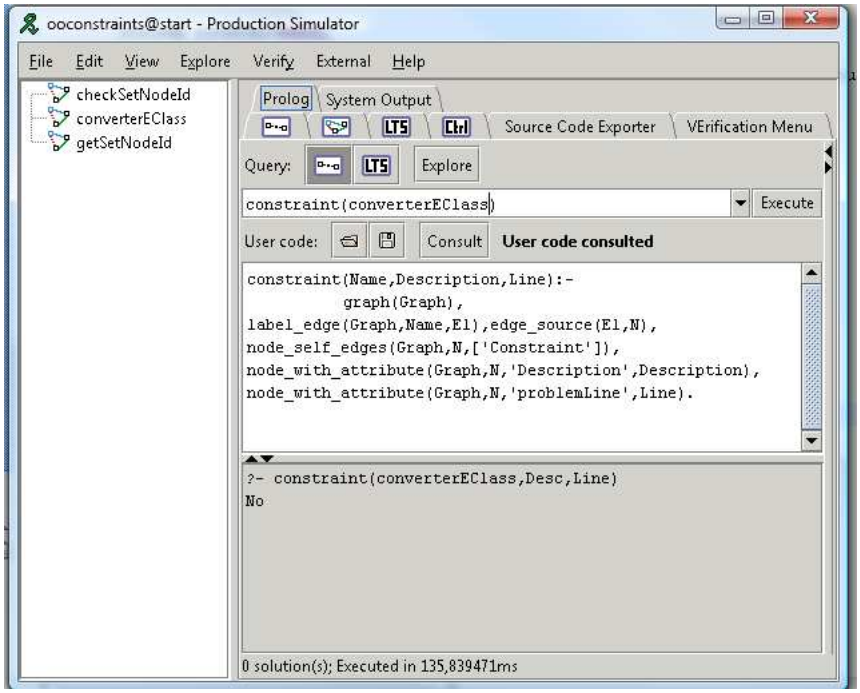


Figure 5.11: GROOVE simulator with Design Query extension: the user has queried for a violation of the constraints for the converter classes.

the query has returned *No*.

Performance Tests

To asses the scalability of the CACV approach, we conducted performance experiments on the 3 versions of the ECORE to GXL converter tool: the first version with 13 converter classes, the current version with 24 converter classes and a median version with 18 converter classes. For each version of the tool, we tested the time it takes to evaluate the constraint *constraintEClass*. We also duplicated each version and modified a class in these duplicates such that this class violates the constraint *constraintEClass*. The performance experiments are conducted on the original and the modified versions of the ECORE to GXL converter tool. GROOVE is executed 100 times on each version and the time it takes run the simulation (i.e. the transformation rules are automatically applied) is recorded for each execution.

Table 5.1, presents the results the of this experiment. Here, the column *No Violation* represents the original version of the ECORE to GXL converter tool and the column *Violation* represents the modified versions (i.e. the modified versions that violate the constraint *converterEClass*). As can be seen from the performance experiments, the

Table 5.1: The execution time of GROOVE for the evaluation of the constraint *converterEClass* on increasing input size. The experiment is conducted on a computer with 2.4Ghz dual-core CPU 4GB Ram running Windows Vista Ultimate (64bit) with JDK 1.6 Update 6

Version	# of Graph Elements	Simulation time (in seconds)	
		No Violation	Violation
First Version	2446	0.35	1.6
Median Version	2975	0.46	3.7
Current Version	3594	0.77	7.8

time it takes to verify when there is no constraint violation is below 1 second. This is because no rewriting is done on the host graph. The time it takes to detect the violation (i.e. apply the transformation rule of constraint) is less than 10 seconds. From these, we can conclude that detection of constraint violations with graph transformation rules is feasible for practical inputs.

5.3.2 Gradient Amplifier Control Software

In this section, we first provide examples on how our approach addresses the drawbacks of the constraint checking tools and then show how constraints remained valid through software evolution.

Expressivity

Usually, industrial software systems have constraints for increasing the maintainability of the source code. As a result, it is important to verify that the developers satisfy these constraints. One example of such a constraint from the control software is using comment blocks to separate different consecutive initializations of a structure with the comment line "//–". For example, the general status registers of the amplifier are represented by a structure called *GeneralStatus* in the software. An amplifier has 14 general status registers; so, the control software has a global array of 14 elements whose type is *GeneralStatus*. In an initialization function, this array is filled; each element's initialization consists of 5 assignment statements. Obviously, when these initializations are placed consecutively it is hard to locate a register's initialization; so, the developers separate them with comment lines.

The AST of the source code is different than the actual source code the developers sees. Due to this, constraints over some program elements, like comments, cannot

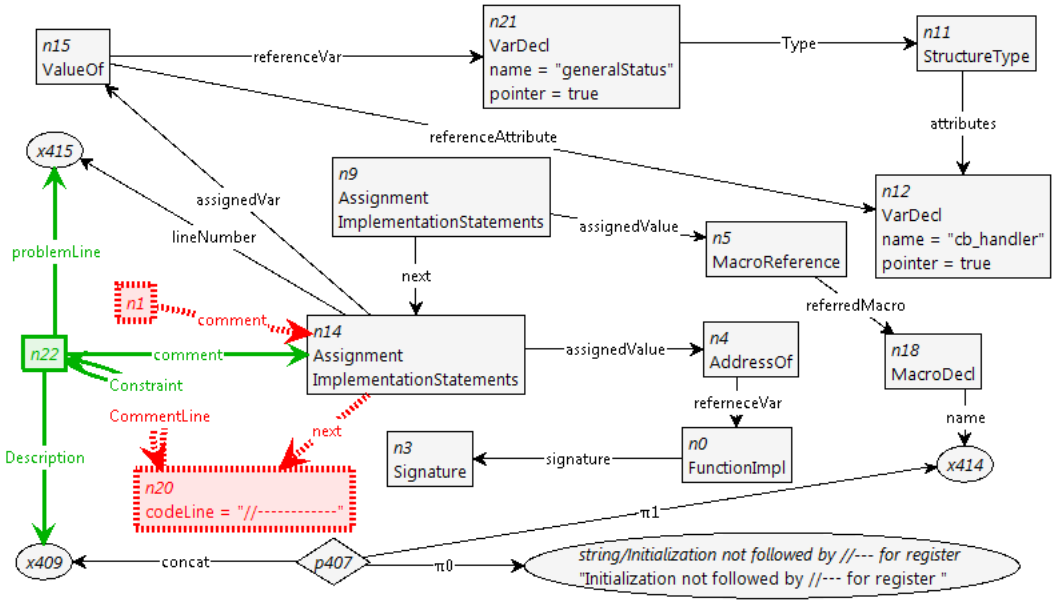


Figure 5.12: The code convention that checks for the comment block after the initialization of the structures for a register.

be expressed. Thus, the constraint of using the comment block to separate initializations cannot be expressed. SCML is designed to represent the source code elements; so, it is possible to express constraints with respect to comments.

Figure 5.12 presents the transformation rule that checks whether the initialization of the structure *GeneralStatus* is followed by the comment block. Here, the node $n14$ is the last assignment statement used for filling the structure. This node is followed by a comment line, node $n13$, whose attribute *codeLine* is set to the comment "//—". The comment line node is a negative application condition of the rule; thus, this rule only matches when the last assignment statement is not followed by the comment block.

Comprehensibility

The approaches to constraint checking in the literature convert all nodes and edges of the abstract syntax tree to facts of Prolog-like languages. In Figure 5.13 the Prolog rule at the AST level used for verifying line 3 of Figure 5.1 is presented. First of all, the developer of the control software could not understand what this constraint is verifying because there is no reference to the macro *GEN_SEOS* and the macro *GENERIC_DUR*. Secondly, it is very hard to understand the portion of the syntax tree the rule is verifying. Obviously, the developers would better understand what

```
variable(device_ptr,DPID),variable(cs_ptr,CPID),value(3000,VID),
variable(GENERAL_STATI,GSID),variable(device_request,DREQID),
valueOf(DPIP,DREQID,LID),valueOf(GSID,RID), assign(LID,RID),
...
variable(dur,DURID),not(valueOf(DPID,DURID,DURASSIGNID), assign(DURASSIGNID,VID))
```

Figure 5.13: The constraint for verifying line 3 of Figure 5.1 expressed in Prolog at the AST level.

the rule is verifying when the constraint visualizes the syntax tree with program elements the developers are familiar with.

A SCML model can be thought of as a syntax tree representing the source code seen by the developers. Figure 5.14 presents the constraint given above as a graph transformation rule in SCML. Here, the macros that the developer is familiar with are used which makes it easier for the developer to understand the meaning of the constraint.

Error Reporting

It is important to provide guidelines to aid the developers in locating the violation at the source code when a constraint violation is detected. Because of the differences between the abstract syntax and the source code, the guidelines provided by the approaches that work at the AST level may be hard to locate at the source code.

SCML preserves the location of the statements with the attribute named *lineNumber*. Thus, besides the names of the program elements and a description text, in our approach error reports also include the line number of the statement(s) where the violation has occurred. As discussed in Section 5.1, the duration for the sector generation request *GENERAL_STATI* should be *GENERIC_DUR*; any duration other than *GENERIC_DUR* is a violation of this constraint. Assume that the reference to macro *GEN_SEOS* with request *GENERAL_STATI* is used with a duration value other than *GENERIC_DUR* at line 1045 of a source file from the control software. Then, the transformation rule shown in Figure 5.7 matches and adds the constraint node where the attribute *Description* is *GENERAL_STATI request used with wrong duration* and the attribute *problemLine* is *1045*. The developer can locate this constraint node and learn that she/he used the wrong duration at line *1045*.

For the example given in the above paragraph, the developer and other stakeholders can also use the querying mechanism and query whether she/he has violated the

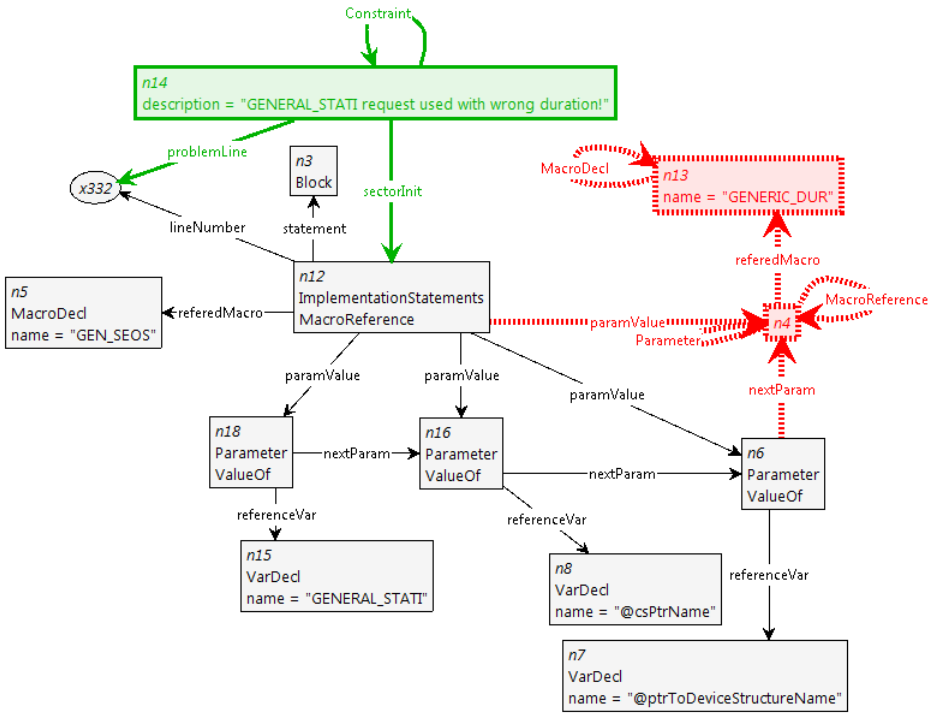


Figure 5.14: The graph transformation rule modeling the constraint that, when a sector is initialized with the request *GENERAL_STATI*, then the duration should be set to *GENERIC_DUR*

constraint *sectorInit* by typing *constraint('sectorInit')* into the *Constraint Query* panel. Then, the query returns:

Line = 1045

Description = GENERAL_STATI request used with wrong duration

Constraints and Software Evolution

We applied CACV to the three different implementations of the gradient amplifier control software to see how many of the constraints remained valid through evolution. From the third implementation of the control software for the latest amplifier model, we identified 10 frequently used constraints with the developers, 7 program constraints and 3 coding conventions (coding conventions are used 87 times in total and the 7 program constraints are used 137 times in total in the latest implementation). The constraints are pointed out to us by the developers in the source code. We modeled these constraints as we see them in the source code and used the tool to verify whether these constraints are violated or not (in the modeling phase the developers were not involved). It may also be possible that some constraints are only in the previous implementations of the control software or only at the third implementation of the control software. To identify such constraints, we manually searched for constraints at the previous two implementations.

The tool showed no constraint violations in the second implementation (the verification took less than a second) and only one constraint violation in the first implementation (the verification took 5 seconds). Manual search for constraints revealed that the code segment for one of the identified 10 constraints is not used in the first implementation but used in the second and the third implementations of the control software. The code conventions are the same over all versions of the control software. This analysis shows that constraints indeed need to be respected during evolution.

5.4 Related Work

The CACV approach converts the source code to an SCML model on which the constraints can be checked. There are approaches in the literature, where the program constraints can be checked within the domain of the programming language. These approaches can be split into three groups. The approaches in the first group are used for detecting the structural constraints. JQuery [95] is an approach based on predicate logic where the structure of the Java programs is transferred to Prolog facts and the structural constraints are expressed as Prolog facts. The *Java Tools*

Language [36] (JTL) is a declarative language designed for selecting Java structural program elements and its semantics are based on predicate logic. Compared to our approach these approaches are limited because they only check the structural constraints and they ignore the implementation of the methods. Thus, the example constraints on the usage of the macro *GEN_SEOS* presented in this chapter cannot be checked with these approaches.

The second group of approaches are the approaches based on AST querying. The examples of these approaches include CodeQuest [65] for Java, which is based on a subset of Prolog and ASTLOG [39] for C/C++. Both approaches work on the AST level and thus constraints on program elements like comments and macros cannot be expressed or checked. Our observations with the industry have shown that macros are used very frequently and usually the developers do not know the implementation of the macro. As a result, they cannot write queries to search for constraints that use macros. Moreover, complex industrial software is a combination of different languages and using different tools to check for program constraints of different languages complicates the procedure as the developers have to understand both tools. This led to the development of SCML, with which can be used to represent both object-oriented and structural languages (i.e. it abstracts the syntax) and which contains elements from the source code. We presented an example where SCML is used to check whether the developer has used the comment blocks correctly.

Eichberg et al. [47] propose an approach for checking for dependency constraints, where the program elements are grouped into *ensembles* and the dependency constraints are defined between the ensembles. This approach converts a subset of AST to predicates; this subset contains declarations and implementation statements that can cause dependencies between program elements. This approach also has two tiers: at the bottom level the core program elements and at the top tier the dependency constraints between the groups of the elements are expressed. The main difference is that our approach is not limited to checking dependency constraints. In the first tier of our approach, the constraints over program elements are expressed and in the second tier these constraints are queried.

In the third group of approaches extensions to type systems are used. The current type checking of programming languages is too limited; it misses the constraints. Bracha et al. [23] proposed pluggable type systems, extensions to type checkers that include constraints. Andreae et al. [14] implement a plugabble type system for Java AST. This system uses Java annotations and declarative rules for defining semantics of these annotations. Such an approach cannot be used in the industrial application presented in Section 5.3.2 because the approach relies the annotations to be in the source code. In the industrial application, the developer implements a new control software by reusing (and adapting) one of the old implementations.

Thus, the developer has to implement the software entities that are to be annotated. With our approach, however, annotations are not needed.

Once the source code is converted to an SCML model, the graph transformations are used for detecting the constraint violations in our approach. The SCML models are attributed graphs so the transition from model to graph is straight-forward. It may be argued that in the model domain (i.e. without transition to graphs) the constraints can be checked using model transformations. When a constraint violation is detected in our approach, the model is updated with information about the violated constraint. For model updates, graph transformations are more suitable than model transformations [62]. Also, in the literature the literature there are many applications of graph transformations to pattern detection. For example, Niere et al. [104] use graph transformation rules to detect and recover the design patterns from source code semi-automatically. Mens et al. [100] use transformation rules to detect inconsistencies between UML models. In these approaches, the right hand-side of the transformation rule is used for adding nodes/edges for marking which nodes/edges match to the pattern the rule has detected. Due to this wide usage and the suitability of graph transformations to model updates, we used graph transformations in our approach. In addition to marking the nodes, we take advantage of attribute operations to extract information useable for guiding the developers to the locations of the constraint violations.

In the context of design patterns, using predicate logic to formally express them is proposed by Eden [45] and Deitrich et al. [42] use it to detect design patterns. These approaches are similar to AST based constraint violation detections because they work at the AST level. Thus, they share the problem of expressivity, comprehensibility and error reporting. Blanc et al. [22] use Prolog rules to express the inconsistencies between UML models. In this approach, the UML models are converted to predicates similar to the AST querying approaches. Since UML is at a higher level of abstraction then the source code the constraints presented in this chapter cannot be expressed. On the contrary, UML class diagrams can be represented in SCML and, then, SCML can be used for checking whether a software system satisfies the class diagram.

5.5 Conclusions and Future Work

This chapter presents a process for computer-aided verification of static program constraints and coding conventions. The main contribution of this process is using meta-modeling for representing the source files. We created a meta-model for representing the source files as seen by the developers; we called this meta-model source

code modeling language (SCML). In this way, constraints over program elements such as macros and comments can be expressed.

Graph transformations are used for detecting the violations of the constraints. The right hand-side of the rule is used for extracting information from the model of the source code that can provide guidelines to the developers about the problem. An important aspect of the transformation rules in our approach is that the names of the program elements are parameterized when modeling the transformation rules. The names of variables may change, for example, from implementation to implementation. The constraints over this variable stay the same; so, rather then modeling different constraints for each name, the value to be checked is parameterized.

It may be hard to manage the constraints of a large software system. To address this problem, we utilize a repository where the constraints are stored in a central storage managed by a repository manager tool created by us. The developers can associate software components with constraints, so, rather than specifying the constraints the developer specifies the component she/he wants to verify. The repository manager fetches the constraints associated for the specified components from the repository. The repository manager also asks the developer to specify the model of the source code (the XMI file). With the fetched transformation rules and the XMI file, the manager launches GROOVE. In GROOVE, if a graph transformation rule matches to the model of the source code, then that the constraint modeled by that transformation rule is violated. When we know the constraint is violated, we need guidelines of the location of the problem. The SCML meta-model elements contain the *lineNumber* attribute which holds the physical line number of the statement. The right hand-side of the transformation rules add nodes labeled *Constraint* which have attributes describing the constraint violation; the rules also combine this description with the line number of the statements where there can be a problem. Thus, the developer can learn about the constraint violation from these nodes.

The model of a software component may contain many nodes and edges; thus, it may be hard for the users to locate the nodes labeled *constraint*. We combined Prolog and GROOVE, so that the developer can enter Prolog queries to find constraint violations. In this way, it is also possible to express *high-level* constraints (i.e. constraints that hide information) or express constraints that are a combination of other constraints using Prolog rules.

We applied our approach to two software systems: one in Java and one in C. In both cases our approach proved to be useful in detecting constraint violations. Moreover, we tested the approach using the history of an industrial software system; we indeed found that it is important to check for constraint violations during evolution.

We programmed dedicated source-to-SCML converters for Java and C. However,

SCML can support other languages such as UML class diagrams and C#. Adding a support for these languages requires one to implement a dedicated converter. With a proof-of-concept Java-to-SCML implementation, we have shown that general purpose text-to-model converters can be used. The benefit of using such a converter is that by only changing the grammar, one can automatically generate the source-to-SCML converter for different languages. The problem we faced while using the general purpose text-to-model converter is the support of scoping. One area we plan to work is to add direct scoping support to the text-to-model tools we employ. Currently, scoping is achieved by late binding where during the load of model to GROOVE the model is traversed to find the variables referred by the statements. We are also developing tools to integrate our approach with Eclipse and Visual Studio, so that constraint violations can be checked in real-time during development.

Chapter 6

Computer-Supported Design Idiom Verification

To add new features and to adapt the software to the changing environment, software systems evolve [90]. Through collaborations with the industry, we observed that to implement change requests the developers tend to use software structures that they are familiar with. In the literature, these structures are termed *design idioms* [116]. A design idiom has a *work-flow*; a step-by-step guide to implement the idiom. The steps in the work-flow specify the *variants* and the *invariants*. Here, the *invariants* are the statements that are imposed by the design idiom. In order to be able to implement these invariants, the constraints they require on other program entities (i.e. program entities they depend on) like type constraints should be satisfied; these entities are the *preconditions* of the invariants. Currently, the applicability of a design idiom to a change request is tested manually; the developers try to implement the change request using the idiom. Our aim is to provide a process (and tools) for computer-aided verification of the applicability of a design idiom to a change request.

In the literature, program transformations are proposed to introduce changes correctly and automatically to the software [18, 115, 121, 120, 122]. These approaches define transformations at the abstract syntax tree level. Similarly, refactoring transformations, transformations proposed for improving the structure, are restricted to a language or also work with AST [105, 106, 99]. The following problems hampered the usability of these approaches in computer-aided verification of design idioms:

1. The preconditions of the design idioms may be program elements that are not represented at the AST, like macro directives.

2. The lack of visualization makes it hard to comprehend and express the steps of a work-flow.

3. Complex software systems are implemented in more than one language.

The contributions of this chapter a process and a tool set for providing computer-aided verification of usability of design idioms; we call this approach computer-aided design idiom verification (CDIV). We address the first problem by using the source code modeling language (SCML, see Section 5.2.1) whose models are used for representing the source code seen by the developers (i.e. includes elements such as comments and macros). The steps of a work-flow in CDIV are transformation rules modeled using SCML elements that check the preconditions and if all preconditions are satisfied, they add the invariant statements. Because SCML has support for program elements that is visible to the developers, rules can be used to check/add any of these program elements.

The models of SCML are attributed graphs and due to maturity of graph transformation engines, the rules are modeled as graph transformation rules. We GROOVE as the transformation engine and developed tools for converting C and Java source files to attributed graphs in SCML. Using GROOVEs editor the transformation rules can be visually specified and the graphs of the source code can be visualized, which addresses the second problem. Because the names of the program elements like variables can change at different implementations of the idiom, CDIV allows these names to be *parameterized*.

The SCML meta-model can be used to represent languages such as C, C++ and Java (C# without attributes) so that rules working on more than one language can be expressed. The benefit of the meta-modeling and generic a transformation engine (like graph transformation engine) is that the meta-model can be extended and rules using these extensions can be expressed without modifying the transformation engine. With this in mind, the SCML meta-model is designed to be extensible and the tool set for converting source to models in SCML have *hooks* in which converters for the extensions to SCML can be programmed.

The semantics of the work-flow can be modeled using a state machine, where the transitions are the transformation rules. The GROOVE graph production tool [111], has a control language [110] that allows one to specify a state machines (where the transitions are graph transformation rules) that restricts the graph transformation engine to apply transformation rules according to this state machine.

The tool set of CDIV contains a repository manager for storing and managing the work-flow and its associated transformation rules. To test the applicability of a design idiom, the developer initiates the process by stating the design idiom and

provides the source code. The repository manager fetches the idiom and the associated rules from the repository. If the rules contain parameters, then the repository manager asks to developer to provide the actual names of the software entities. With the provided names, the transformation rules are *bound* and after this binding, the repository manager launches source-to-model converter. Finally, the work-flow is simulated on the model of the source code with GROOVE. If an invariant cannot be implemented, then the design idiom cannot be implemented as is (i.e. without modifying the structure of the software) and GROOVE reports the step that has failed. If all steps in the work-flow of a design idiom can be applied, then the invariants of the design idiom are not violated. At this stage, GROOVE launches model-to-source code converter to generate the source code which correctly implements the invariants of the design idiom. The developer can takes this code and implement the variants.

This chapter is organized as follows: the next section describes the problem and provides an example from an industrial software system that is used throughout the chapter. Section 6.2 details the process. The graph-based model that is used for representing source code, modeling the invariants using graph transformations and modeling the work-flow specifications is detailed in Section 6.3. The details of the computer-aided verification and the template source code generation are provided in Section 6.4. Section 6.5 shows the application of our approach to an open source and an industrial software system. Section 6.6 discusses the related work and, finally, section 6.7 provides the conclusion and the future work.

6.1 Motivating Example

The change of the gradient amplifier of a Magnetic Resonance Imaging (MRI) scanner is an anticipated evolution that happens on average every two years. The MRI software controls the gradient amplifier through an interface of functions. The control software of an amplifier must implement this interface.

The control software of each amplifier uses a driver that communicates with the amplifier. To read a value from a register at the amplifier, the driver provides a function that takes an functional pointer, a time value and the address of the register. At the given time, if a value is ready on the register, the driver calls the call-back function and passes the read value.

One of the important function of the interface is getting the status of the amplifier, so that it is operated under right conditions. Every amplifier has a set of status registers that store data about that amplifier's status like the current temperature

of the amplifier. However, each amplifier model has its own set of status registers; that is, from model to model the addresses of the registers can change, some status registers can disappear and new ones can be added.

There are currently three versions of the control software controlling different amplifier models. In each version the requirement of reading a status register is implemented as follows:

1 Implement a structure called *General Status* structure that holds attributes for holding the register address and the call-back function, if it does not already exist.

2 Implement the function where the status structure for each status register is initialized, if it does not already exist.

3 Add the variable that holds the value read from the amplifier register, if it does not already exist. Note that two or more registers can use the same variable.

4 Add the call-back function that stores the value read from the amplifier or the *overridden* value in the variable added in the previous step, if it does not already exist. This step it is also possible to use the same call-back function for two or more registers.

5 Fill the general status structure entities for the register at the function implemented in the previous step.

6 Implement the function that registers the call-back functions with the driver, if it does not already exist.

7 Add the function call that instructs the driver to read the value of the status register in the function implemented at the previous step.

8 Add the function that returns the value of the variable added in step *6* after some conversions. This function is used by the upper layers of the software.

For each status register the developers implement this work-flow. From the work-flow only step *5* is required by the communication driver and only the last step is required by the gradient control interface. The other steps are imposed by the design idiom.

Figure 6.1 presents the invariant code segment for step *4* for two different amplifiers, *Amplifier1* and *Amplifier2* (note that this a stripped down implementation and the names are not realistic). As can be seen, the implementation of the invariant is similar for both amplifiers. To be able to implement this code segment, the developer has to make sure that the *AmplifierXStatus* structure has the right fields (with the right types), the call-back function has the right signature and the right address. Here, the register address are defined using preprocessor directives,

```
Amplifier1Status *generalStati;
generalStati=&device.status[WATERCOOLER];
generalStati→cb=Amplifier1_Cooler;
generalStati→regAddress=WCOOLER_REG;
//————————————
```

```
Amplifier2Status *generalStati;
generalStati=&device.status[FAN];
generalStati→cb=Amplifier2_Cooler;
generalStati→regAddress=FAN_REGISTER;
//————————————
```

Figure 6.1: The invariant code used for registering a call-back function of a status register to the real-time interface driver. For different amplifiers the register address and the call-back function change but the registration procedure does not change.

like *FAN_REGISTER*, in the amplifier driver. The developer uses these macros in developing a control software for an amplifier.

For a new amplifier, the domain expert tries to apply this design idiom. However, manually checking all the preconditions of the invariants is time consuming. Moreover, the design idioms are usually not documented; thus, to use an idiom, unfamiliar developers copy and adapt the old implementation of the idiom. This in turn causes bugs to be introduced due to missed preconditions. The rest of the chapter describes an approach to automate the verification whether an invariant can be used or not. The example of registering the call-back functions presented in this section, is used as a running example in describing our approach.

6.2 A Process for Computer-Aided Design Idiom Verification

Usually, to implement a design idiom, the developers follow a work-flow. In each step of this work-flow they implement the *invariant* and the *variant* statements. The *variant* statements are imposed by the requirement the developers are implementing. The *invariant* statements, on the other hand, are crucial to the correct operation of the design idiom. The invariant statements have preconditions that should be satisfied. In CDIV, graph transformation rules verify whether preconditions of the invariants of a step are present in the source code or not. With the work-flow model, the rules are put into the order the developer follows when implementing the design idiom. In this way, the dependencies between steps are resolved and the feedback

about the failing step is presented the developer can easily recognize the failing step. Besides ordering, two other important aspects of the work-flow are capturing the conditions and the alternatives. The developers that do not know these may introduces bugs to the software.

CDIV involves three actors, namely the language engineer, who knows how to model the steps of a work-flow as graph transformation rules in CDIV and the work-flow, the developer and the domain expert, who is a developer but has experience in the design idiom. The process steps are divided into manual preparation and computer-aided steps. The steps that are done manually involves modeling the design idiom:

1. The domain expert describes the work-flow and the invariants of design idiom.
2. The language engineer specifies the work-flow in GROOVE's control language (we modified this language to support parametrization) and models the invariants as graph transformation rules in SCML. In our previous study [32], we used graph transformations to model the evolution of design patterns. We programmed a tool called Computer-Aided Design Evolver (CDE) to store these transformation rules in a repository. For the computer-aided design idiom verification process, we also use the CDE tool store the work-flow and the graph transformation rules in a repository.

The steps that are computer-aided start with the domain expert or the developer specifying a the design idioms she/he wants to test. After this specification the process is executed as follows:

1. The CDE tool fetches the work-flow from the repository. It parses the work-flow and fetches the transformation rules that are referred in this work-flow.
   (a) If the transformation rules have parameters CDE tool asks the developer provide the actual names.
   (b) The CDE tool replaces the parameters in the transformation rules with the provided names. This step is called *binding*.
2. The source code is converted into a model in SCML. We programmed dedicated source-to-SCML converter for Java and C (can parse C++ class declarations). We also used implemented SCML meta-model in KM3 [74] and a proof-of-concept Java grammar in TCS [75] to show that general purpose source-to-model converter generators can also be used to generate a source-to-SCML and SCML-to-source converter.
3. Using the developer provided values for the parameters, the CDE tool binds the graph transformation rules.

4. CDE launches GROOVE.

5. The GROOVE simulates the work-flows. We extended GROOVE with a state machine tracer that compares the input work-flow with the transitions the simulator applied. In case a step in the work-flow cannot be applied, the developer is notified with the step that has failed. If all the steps in the work-flow are applied then template source code is generated.

In the remaining sections of this chapter, we describe how the process is realized using graph semantics by the tools.

6.3 Design Idiom Modeling

Design idioms consist of invariants that are implemented according to a work-flow. Automated verification of whether a design idiom can be used or not requires these to be explicitly modeled. This section describes how we model the work-flow and the invariants in a step of a work-flow.

6.3.1 The Source Code Modeling Language (SCML)

In CDIV, we use the SCML because it is designed to represent the source code the developers see and work-on. For example, it is possible to express rules that depend on macros with SCML. In section 5.2.2, the meta-model of SCML is explained; however, in this chapter the SCML is explained through an example to increase the readability. Figure 6.2 depicts the lines 9-10 of Figure 6.1. Here, these statements are connected to a block statement (the node labeled *block*) which represents the body of a function implementation. This is because the block statement is connected to a function implementation node with an edge labeled *body*. This function implementation's signature (derived by following the edge labeled *signature*) is the signature named *init_structs*. From this, we understand that these statements belong to the *init_structs* function. The function *init_structs*, as depicted in the figure, has also a variable declaration statement with the name *genralStati*, which is a pointer of the structure type *Amplifier2Status*. The assignment statement, line 9 of Figure 6.1, is the emphasized node. The left-hand side of the assignment, modeled by an edge labeled *assignedValue*, is connected to a *MacroReference* statement. Following the edge labeled *referedMacro*, it can be seen that the referred macro is the macro named *FAN_REGISTER*. As a result, the assignment assigns the value of this macro.

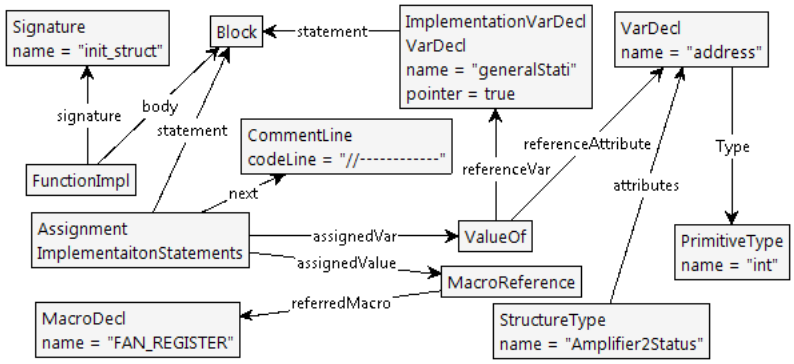


Figure 6.2: The graph base representation of the statement *generalStati* → *regAddress* = *FAN_REGISTER and comment block* //———-.

The variable that gets the value of the macro *FAN_REGISTER* is the attribute *generalStati* − > *regAddress*, because the right-hand side of the assignment statement, designated with the edge labeled *assignedVar*, is a *ValueOf* statement. The reference variable of the *ValueOf* statement (modeled by the edge labeled *referenceVar*) is the pointer *generalStati* and the reference attribute (the edge labeled *referenceAttribute*) is the variable *regAddress*, which is an attribute of the structure *Amplifier2Status*.

The comment block in Figure 6.1 serves an important purpose for maintainability: it differentiates different instantiations of the general status structure. There are on average 14 such instantiations for an amplifier; thus, the code becomes very hard to read without this comment block at the end of each instantiation. In SCML, single comment lines are represented with nodes labeled *CommentLine*. The attribute *codeLine* of these statements are set to the comment in the source code. For example, the comment at line 10 in Figure 6.1 is represented in Figure 6.2 with the node labeled *CommentLine* whose attribute *codeLine* is //—-.

6.3.2 Modeling the Invariants of a Work-flow Step with Graph Transformation Rules

A step of a work-flow contains variants and invariants. As discussed before, the variants are imposed by the requirement the developer is implementing and the invariants are imposed by the design idiom. The CDIV process aims at verifying whether the steps of the design idiom can be implemented. This involves verifying wether the preconditions of the invariants are satisfied or not. Due to this, the model of a work-flow step only consists of the invariants. In detail, the model of a work-flow step includes an algorithm, preconditions and parameters. The parameters are

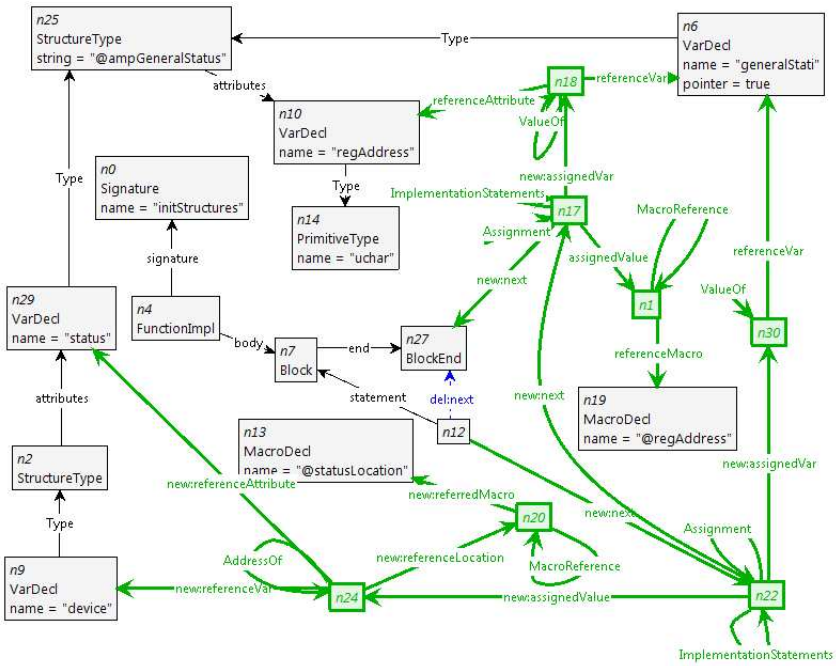


Figure 6.3: The graph transformation form of the invariant *initStatusStructs*.

the names of the program entities like variables that can/may change for different implementations of the idiom. The algorithm describes how the invariants are added to the source code.

Step 5 of the design idiom presented in Section 6.1 is filling the structure general status for a register with the right call-back function. Even though the structure general status differs in different amplifier models, each structure has an attribute that holds the address of the call-back function. The algorithm of this step, which we call *registerCallBack*, adds the lines 2, 3 & 4 in Figure 6.1. Looking at the two different implementations of this invariant, it can be seen that the name of the amplifier's status structure, the array location of the general status handler in the amplifier's device structure and the name of the call-back function changes; thus, these are the parameters of the invariant. In order to correctly add these lines, the following conditions should be met: the call-back function should return a boolean, a variable named *generalStati* with the right type should be declared and the structure general status of an amplifier should have an attribute labeled *cb_handler* which is a functional pointer.

A work-flow step is executed by first checking for the preconditions and, then, if all preconditions are satisfied, adding/removing the statements specified in the algorithm. A precondition of a work-flow step can be thought of a pattern that should

exist in the SCML model of the source code. If this pattern does not exists then the precondition is violated and the algorithm of the work-flow cannot be executed. This execution is very similar to graph transformations. Informally, a graph transformation has a left-hand side that specifies what should be in the host graph for the rule to match and a right-hand side that replaces the left-hand side when the rule is applied (interested readers on the subject are referred to the literature [46]). In GROOVE both left-hand and right-hand side are shown in the same graph: the nodes/edges that green and that are tagged with the keyword *new:* are going to be added when the rule is applied, the dashed blue nodes/edges , which are tagged with the keyword *del:*, are going to be deleted and the red thick dashed nodes/edges, which are tagged with the keyword *not:*, are the negative application conditions [68]. The rest should be in the graph in order for the rule to match.

The algorithm and the preconditions are modeled as graph transformation rules by following the SCML meta-model (Figure 5.2). The left-hand side of these transformation rules include the preconditions of the invariants (of a work-flow step) and the right-hand side is the algorithm. Figure 6.3 presents the graph transformation rule modeling the invariant *initStatusStructs* (only lines 2 & 4 are shown). The algorithm of this invariant adds three assignment statements, so in the graph transformation rule these statements are the nodes that are added. The rest of the nodes are the preconditions of this invariant. For example, the precondition that the amplifier general status structure should have one unsigned character (*uchar*) attribute named *regAddress* is modeled by a variable declaration node (node *n28*) that is connected to a structure node (node *n30*). If this attribute is missing or its type is not *uchar* then the transformation rule does not match, thus the invariant cannot be added.

Although not shown in Figure 6.1, the assignment statements mentioned above are added to a function named *init_structs* and they are implemented before end of the body of this function. The block node *n7* is the start of the body of this function and the block end node *n27* is the end of this function's body. The transformation rule deletes the edge labeled *next* between the nodes *n12* and *n27* and adds two assignment statement between these nodes; the added assignment statements are nodes *n22* and *n17*. Node *n12* is a generic node; it can match to a node representing any statement. Thus, whatever the statement comes before the end of the body block of the function *init_structs*, the rule matches (provided that the preconditions match) and adds the assignment statements. In this way, for example, the same invariant can be used to add these statements for different registers.

Looking at the implementation shown in Figure 6.1, it can be seen that the names of the amplifier structures change (e.g. Amplifer1GeneralStatus and Amplifier2-GeneralStatus). The design idiom has a precondition the attribute *regAddress* of

these structures. So, we need to check whether the amplifier structure has an attribute named *regAddress* before adding the invariant statements. However, we need to be able verify this for the amplifier structure we are currently working on. This can be achieved by changing the name attribute of the structure type node representing the amplifier structure (node $n25$) in the transformation rule. For example, if this name is set to *Amplifier1GeneralStatus* then the rule can verify the precondition on the *Amplifier1GeneralStatus*. By changing the name to *Amplifier2GeneralStatus* the same rule can be used to verify the precondition on *Amplifier2GeneralStatus*. In CDIV, such differences on the names of the program entities are achieved with parameters of the work-flow step. The name of a program entity that is parameterized is modeled by setting the attribute *name* to @ < *parameter_name* >. In the graph transformation shown in Figure 6.1, the name of the amplifier structure is parameterized by setting it to *@amplifierGeneralStatus*. In order execute this work-flow step (or to apply this transformation rule), the values of these parameters should be supplied; for example, the name of the amplifier structure should be given for a new amplifier. Replacing the parameters by the actual names of the software entities is called *binding*.

6.3.3 Design Idiom Work-Flow Modeling

The work-flow can be modeled using state machines where the states represent the source code and the transitions are the steps. We model the semantics of the invariants of a work-flow step using graph transformation rules. Thus, the transitions of the state machine modeling the work-flow correspond to the graph transformation rules that should be applied to the given state of the source code (i.e. the host graph modeling the source code).

Algorithm 2 The work-flow used for registering a call-back function with the amplifier driver.

```
1: function initGeneralStatus(statusResigterName) {
2: try{
3: addInitFunction_statusRegisterName;
4: }
5: addCallBackFunction_statusRegisterName;
6: initGeneralStatusStruct_statusRegisterName;
7: }
```

GROOVE is a state-space generator where each state is a graph and the transitions are the applications of the graph transformation rules. The inputs to GROOVE are a graph production system (a set of graph transformation rules) and an initial graph.

The initial graph is also the initial state of the state-space. At a state GROOVE automatically finds the graph transformation rules that can be applied; applying a graph transformation rule changes the state.

GROOVE's control language [110] is used for specifying state machines whose transitions are transformation rules. The specified state machines are used the restrict the transformation engine; at each state the transformation engine is restricted to apply only those rules which are specified by the state machine. We express the work-flow of design idioms in the control language. GROOVE's transformation engine is restricted to follow the work-flow for a given design idiom.

The repository manager (a tool programmed by us) is responsible for locating/storing the work-flows and the associated transformation rules. The work-flows of the idioms is stored in a repository as functions in GROOVE's control language. o Algorithm 2 presents the work-flow of the design idiom used for reading the values of the status registers (due to space limitations only 3 steps are presented). Here, the first line is the function declaration; the function declarations of design idioms are of the form $< idiomName > (< InitializationName >)$. Each design idiom is given a unique name; the idiom depicted in Algorithm 2 is named *initGeneralStatus*.

The design idiom name is followed by the *initialization name* in parentheses. A design idiom can be used more then once and the initialization name is used to identify each usage of the idiom. For example, to use the work-flow *initGeneralStatus* for the status registers *boardmodelhigh* and *boardmodellow* the domain expert initializes the work-flow *initGeneralStatus* twice by giving it the following initialization names: *initGeneralStatus(boardmodelhigh)* and *initGeneralStatus(boardmodellow)*.

The steps of the work-flow are expressed as $< stepName >\_ < InitializationName >$. The step name is the name of the graph transformation rule modeling the semantics of the work-flow step. The work-flow in Algorithm 2 uses three transformation rules: *addInitFunction, addCallBackFunction* and *initGeneralStatusStruct*. The initialization name is the same as the initialization name in the function declaration statement. For example, when the domain expert initializes the work-flow to be used for a change request as *initGeneralStatus(boardmodelhigh)*: first the repository manager fetches the transformation rules associated with the work-flow. Then, the transformation rules names are padded with the instance name; so *addInitFunction* becomes *addInitFunction_boardmodelhigh*. Lastly, the tool replaces string *statusRegisterName* (the instance name) in the work-flow function with *boardmodelhigh*. This last step binds the initialized work-flow with the transformation rules.

Figure 6.4 depicts the state machine generated when initialized work-flow code is loaded in GROOVE. Here, the transition *addInitFunction_boardmodalhigh* corresponds to step *2*. This step describes the signature of a function in which the

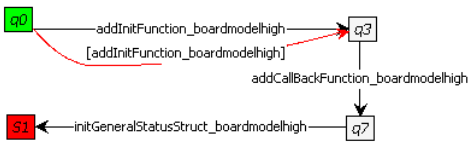


Figure 6.4: The work-flow used for registering a call-back function with the amplifier driver.

structures for the registers are filled. This step should be done only if this function signature is not already implemented. However, the remaining two steps should always be implemented. The graph transformation rule *addInitFunction* is designed so that it matches to the host graph only when the function signature does not exist in the host graph. Thus, we need a way to instruct GROOVE to skip this step if this transformation rule does not match. The *try* statement in line 2 of Algorithm 2 is used for this purpose. The *try* statement causes two transitions to be added between states S_0 and q_3: the first transition labeled *addInitFunciton_boardmodelhigh* occurs when the transformation rule with the same name matches. The second transformation rule labeled *[addInitFunciton_boardmodelhigh]* occurs only when the transformation rule *addInitFunciton_boardmodelhigh* does not match.

At the state q_3, there is only one outgoing transition labeled *addCallBackFunction_boardmodelhigh*; this means that at this state GROOVE can only apply the transformation rule with the same name. If this rule can be applied, then the system can reach the work-flow state q_7. The state q_7 has also one outgoing transition labeled *initGeneralStatusStruct_boardmodelhigh*. Thus, GROOVE can only apply the transformation rule with the same name at this state. If this rule can be applied, the system reaches a final state which means that the work-flow is completed. If one these transitions cannot be applied then the work-flow cannot be completed.

To place a new design idiom in the repository, the graph production system containing the transformation rules and the work-flow is prepared using GROOVE (i.e. using GROOVE's graph and control editor). Then CDE is launched with the location of the new production system. CDE parses the work-flow file to extract the instance name parameter, the name of the idiom. Then, it copies the transformation rule files and the work-flow file into the repository.

6.4 Using The Approach

To use the approach the domain expert/developer specifies the design idioms he/she wants to verify, using a specification language that is very similar to GROOVE's

control language. We modified the Computer-Aided Design Evolver tool [32] (CDE) to manage the repository. Once the design idiom specification is supplied, CDE looks for the work-flow definitions at the repository. CDE extracts the parameters from the transformation rules referred by each work-flow to form the *annotation* file. The annotation file contains all the parameters of the transformation rules the desired work-flows use. The annotation file is filled by the developer and CDE is executed with the filed annotation file. Using the values of the parameters, CDE binds the transformation rule and forms a graph production system that can be loaded in GROOVE to simulate the work-flow.

Using the parser tools, the source code is converted to a graph that follows SCML and together with the production system it is loaded into GROOVE. We added the menu *work-flow verifier* to GROOVE which hides the transformation rules and the control program. Once everything is loaded, this extension to GROOVE automatically applies the work-flow and displays the errors (if any exist). This section describes the design idiom specification, the binding procedure and the verification process. Note that, we also programmed a proof-of-concept Java-to-SCML converter using TCS. If this converter is used, then the resulting model is an XMI file. We extended GROOVE so that these files can be loaded directly from GROOVE.

6.4.1 Specifying the Design Idioms

The domain expert specifies the design idioms she/he wants to test, using GROOVE's control language with minor additions. We have modified CDE to generate the graph production system from this modified language. Algorithm 3 shows an example specification. Here, the first line states where CDE should form the graph production system that contains the work-flow and graph transformation files.

Algorithm 3 An example specification that uses the design idiom *initGeneralStatus* from the repository *addAmp*.

```
1: output evolvemec.gps;
2: useRepository addAmp;
3: idioms{
4: initGeneralStatus(boardmodelhigh);
5: initGeneralStatus(boardmodellow);
6: }
```

The repository is divided into libraries; a library holds the design idioms with its work-flow specification and transformation rules. Libraries are formed by the developers/domain experts; for example, each library can refer to a component of a

software and store the design idioms related to that component. With the statement *usingRepository <RepositoryName>*, the libraries re added to the search path of the CDE tool. The example specification, Algorithm 3, at line 2 states that it uses the repository *addAmp*.

Within braces after the statement *idioms*, the design idioms that are going be verified are placed. We use GROOVE's control language syntax for design idiom specification; thus, anything in between the braces should follow the syntax of the control language. At lines 4 and 5 the example specification presented in Algorithm 3 states that it uses the design idiom *initGeneralStatus* twice: first the one with instance name *boardmodelhigh* is applied and then the second one with the instance name *boardmodellow* is applied. With the given design idiom specifications, CDE fetches their work-flows from the repository, forms the control program to be used by GROOVE and places the program in the output graph production system.

The work-flow of the design idiom *initGeneralStatus* consists of 3 steps (Algorithm 2); since this example specification uses it twice, the output graph production system contains 6 transformation rules. For example, in the output production system the graph transformation rule *initGeneralStatusStruct* (Figure 6.3) occurs twice: one with name *initGeneralStatusStruct_boardmodelhigh* and the other one with name *initGeneralStatusStruct_boardmodellow*.

6.4.2 Binding Invariants to Source Code

When the *CDE* tool generates the graph production system, it also generates an annotation file. This file lists the parameters that are used in the graph transformation rules of the work-flow. The annotation file is a listing of the form *@<parameterName> _ <instanceName> =*. The developer/domain expert replaces the description text with the name of the software entity in the implementation. Note that if a parameter is shared among different instances of design idioms, then the developer can omit the *<instanceName>*.

After filling the annotation file, CDE is executed again and in this execution CDE replaces the parameter values of the attributes with the supplied value. For example, if the developer supplies the annotation *@ampGeneralStatus_boardmodelhigh = Amplifier2GeneralStatus* then CDE replaces the string *@ampGeneralStatus* with *Amplifier2GeneralStatus* in all transformation rules whose names end with *boardmodelhigh*.

The amplifier *Amplifier2* has 14 registers for which a call-back function should be registered and in each registration name of the structure general status is *Am-*

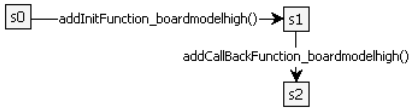


Figure 6.5: Incomplete simulation of the work-flow of Figure 6.4; the last graph transformation rule does not match

plifier2GeneralStatus. So rather then supplying this annotation for each registration, the developer only supplies the annotation *ampGeneralStatus = Amplifier2GeneralStatus*. This instructs CDE to replace the string *ampGeneralStatus* in all transformation rules placed in the output production system.

6.4.3 Work-flow verification

The verification of whether the work-flow is completed or not is done by comparing the state-space generated by the graph production simulation to the work-flow definition. When a simulation is executed, GROOVE generates a state-space where the transitions are the names of the graph transformation rules that are applied. If in this generated state-space, a path from a start state to a final state is also a path to final state in work-flow specification then the work-flow is completed. If not, then there are still steps that need to be applied in order to complete the work-flow. Because the work-flow specification may define more than one final state (e.g. there can be more than one outgoing transition from a state), it is sufficient to find that one of these paths are in the generated state-space.

In Figure 6.5, an example state-space generated from the simulation with the work-flow given in Figure 6.4 is presented. Here, the path from a start state to the final state does not lead to a final state in the work-flow because the transition *initGeneralStatus_boardmodelhigh* is missing in the generated state-space. The work-flow is not completed and the problem is related with the *boardmodelhigh* instance of the work-flow step *initGeneralStatus*. We extended GROOVE with a state-space tracer that executes the verification described above. If the tracer finds that the work-flow is not completed, it lists the steps that are not applied. Figure 6.6 presents a screen shot of GROOVE with the work-flow verification. The work-flow step *initGeneralStatusStruct* with the instance name *statusx* has failed and this is reported to the user.

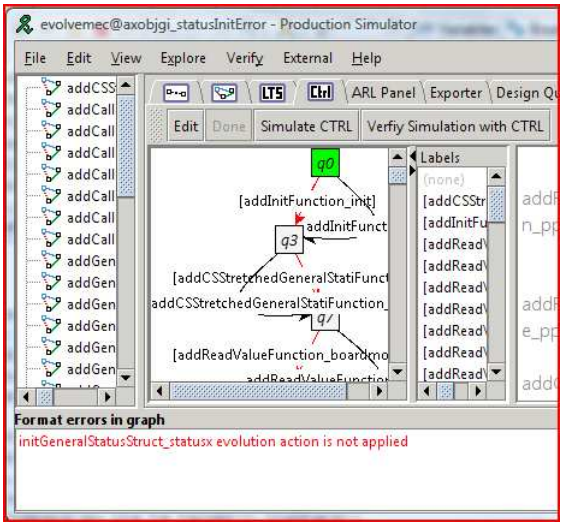


Figure 6.6: Screen shot of GROOVE with work-flow verification extension.

6.4.4 Template Source Code Generation

The source code conversion is done with a separate graph-production system that traverses the host graph. For each statement node in the SCML meta-model there is a graph transformation rule that recognizes the statement. For each transformation rule used for exporting source code there is a matching exporting strategy. These strategies convert the statement nodes to statements in the programming language specified by the user. We extended GROOVE to call the right export strategy, depending on the matched rule (currently, the conversion only works with C++ and Java).

The generic statement nodes are used for representing the statements that the source code to graph converter cannot parse. The attribute *codeLine* for these statements is set to the actual statement read from the source code. In this way, code generation is able to cover the whole source files (i.e. not only the statements that can be represented with the graph-based model).

Algorithm 4 The control function used for traversing the statements of a function

```
1: function exportFunction(){
2: selectFirst;
3: alap{
4: exportCall() | exportVarDecl() | exportAssignment();
5: moveNext;}
6: }
```

The traversing of the host graph is done by following the edges labeled *next* connecting statements. Algorithm 4 and algorithm 5 presents two control functions: the first one is used for traversing the statements of a function and the second one is used for exporting the assignment statement. Here, the transformation rule name *selectFirst* selects the first statement in the function. The statements of function can be either a call, an assignment or a variable declaration. In the control automaton, this is expressed by connecting the control functions used for exporting these statements with *or* (|, line 4 in Algorithm 4). Once the conversion of a statement is finished, the conversion moves to the next statement in the function; the transformation rule *moveNext* is used for this purpose. The traversing of the statements in a function continues until all the statements are traversed. At line 3 the state *alap* stands for *as long as possible* and the transformation rules that it encloses within the curly braces that come after it are repeated until they do not match to the host graph.

Algorithm 5 The control functions used for converting an assignment statement

```
1: function exportAssignment(){
2: selectAssignedVar;
3: exportValueOf() | exportAddressOf();
4: exportAssignmentStatement;
5: selectAssignedValue;
6: exportCall() | exportValueOf() | exportAddressOf() | exportAssignment();
```

When the traversal of statements reaches an assignment statement then the transformation rule named *selectAssignedVar* matches, line 2 in Algorithm 5. The assigned variable of an assignment statement (i.e. the left-hand side) can be a value/address of a statement (Figure 5.4). Thus, the call to the control functions *exportValueOf* and *exportAddressOf* are connected by an *or* condition. After the assigned value is exported, the transformation rule *exportAssigmentStatement* matches, which marks the statement as *exported*; the matching of this rule triggers GROOVE to execute the export strategy *Assignment* that adds an assignment operator to the source code. The rule *selectAssignedValue* marks the right-hand side of the assignment and depending on the kind of the statement the statement is exported.

If the SCML meta-model is extended (and if the resulting source code should include statements from these extensions), the model-to-source converter should also be extended. The extension to model-to-source converter involves modeling a transformation rule that recognizes the statement and implementing the converter strategy for the statement. We are currently implementing save as XMI option to GROOVE so that the model-to-source converter generated by TCS can be used for source code generation purposes. With TCS, the extensions to SCML meta-model only involve

modifying the KM3 file and the grammar file; that is, no specification converter strategy is required.

6.5 Application of the Approach

We applied the approach to the open source and the industrial software presented in Section 5.3.

The gradient amplifier control software consists of several design idioms. From these we picked the 3 idioms that covered the most lines of code for the latest control software. Then we applied the approach to generate the template code for these idioms for the oldest version of the amplifier control software. The goal of this study is to show that the design idioms stay valid through evolution. The findings about this study are presented in next subsection. An important aspect of these idioms is that they depend and add statements from an in-house developed domain specific language (the statements of which are embedded in C++ source files). Because our approach uses an intermediate model to represent the source code rather than the AST, statements of this language can easily be represented with our approach. In the next section, an example work-flow step that has a dependency to this language is also presented in the next subsection.

For the open source ECORE to GXL converter software system, the goal is to show that the verification of the applicability of the design idiom and the actual implementation agree. Besides this, we also show that our approach can be used to enforce constraints that are not possible to enforce using only the source code and AST: The converter uses a meta-model defined with KM3, developer has to follow the meta-model to implement a converter otherwise the tool may crash. To address this problem, we defined graph transformation rules that check the meta-model and if the meta-model agrees, the conversion statements are added. Subsection 6.5.3 presents this study.

6.5.1 Gradient Amplifier Control

The MRI software defines an interface of 32 function to control the gradient amplifier. The communication between the MRI computer and the amplifier is done with two drivers; one communicates over a serial port and the other over a real time port. The control software implements these 32 interface functions and it allows the MRI software to communicate with the gradient amplifier using the drivers. The control software is specific for each amplifier and on average the control software has

4485 lines code written in C with many preprocessor directives. The first version of the control software is implemented in the year 2003 and the latest version is from the year 2007. In this section, first we show two examples of how SCML preserves program elements visible to the developer and then, we show that design idioms are important during evolution by looking at the history of the control software.

Using Program Elements from Source Code

In literature, the proposed approaches for program transformations work use the abstract syntax tree of the program. Although, the transformation rules can be specified using the target language, at the abstract syntax tree program elements such as comments, macros are lost and rules using these elements cannot be specified. This hampers to usability of these approaches for verifying the design idioms because the design idioms are implementations/way-of-workings the developers are familiar to and most of the time a developer is familiar with the source code. So, when we ask developers to specify the preconditions, they use the program elements they see in the source code.

The control software runs in more than process so it is important to ensure synchronization between these processes. The designers defined macros to replace the calls to the semaphores; the developer only needs to specify the entrance and the end of a critical section rather than making calls to initialize the semaphores and locks. The 3 idioms we modeled heavily use these macros. Because at AST level macros do not exist, we can define a Stratego/XT [121] like transformation rule for a step of one these idioms as follows:

addSync: set_analog_section(
gen_device_ptr) $->$ int *sem=getSem(*gen_device_ptr* ,...) semlock(sem)
set_analog_section(*gen_device_ptr*) semunlock(sem)

Note that the function calls are not the actual calls the macros used in the control software replace; these are stripped/renamed version of these calls used here just to provide an example. The developer of the control software have no knowledge of the semaphore library, so she/he could not express this transformation. Even if she/he has expressed, an error can be made in the parameters of the semaphore calls; in fact, the major benefit of using the macros is hiding the details about the functions easing the use of the library.

The developer only uses two macros for defining a critical section; let's assume that these macros are called *PROCEED* and *READY*. Since SCML directly supports macros, a transformation rule using these macros can be specified as shown in Figure 6.7. In the figure, the macros are represented with nodes *n12* and *n1*. The nodes

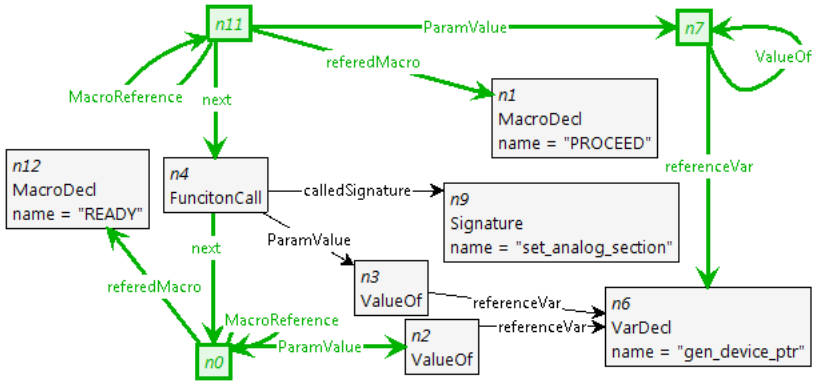


Figure 6.7: The transformation rule that adds the macros used instead of semaphore calls

are at left-hand side of the transformation rule because they are preconditions. The references to the macros are nodes $n11$ and $n0$, which are added by the node. When applied, this rule surrounds the function call *set_analog_section* with the macros used for synchronization. With this rule the benefit of meta-modeling can be seen where the developer can express and generate code with the programming elements they are familiar with.

6.5.2 Representing Code Conventions in CDIV

Usually, the industry has some coding conventions used to increase the maintainability of the source code. It is important for a program transformation approach to generate source code following these conventions. The program transformation approaches work with the AST which does not include most of the code conventions. As a result, these approaches rely on pretty-printers in transforming the AST to source code. This, in turn, can generate source code that does not follow the coding conventions.

The transformation from source-to-SCML, on the other hand, preserves the program entities visible to the developer. For example, comments and macros are directly supported by the meta-model of SCML. For example, the code convention of following each structure initialization with the comment line //—- can be expressed with the program transformations that work on the AST level. The transformation rule shown in Figure 6.3 is able to add this comment line. Moreover, the generic statements (the nodes with label *statement*) can be used for representing program entities not covered by the SCML meta-model. For example, new lines are not covered by the SCML meta-model; however, we programmed the source-to-SCML converters

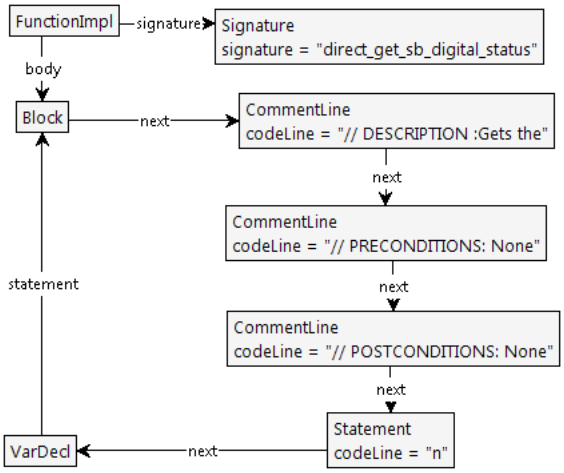


Figure 6.8: A coding conventions with a comment block followed by a new line modeled in SCML

(both Java and C converters) to recognize the new lines and convert them to generic statements with the attribute *codeLine* set to new line.

The control software developers have a coding convention in which after each function block, a description of the function, its pre- and post-conditions are written in comment lines. This comment block is followed by a new line after which the first statement of the function is implemented. Figure 6.8 shows a portion from a function where this coding convention is modeled in SCML. Here, the first statement of the function is variable declaration. The new line which is modeled with a generic statement comes before this variable declaration statement and after the comment line. After a design idiom is simulated, the code generation step preserves this coding convention by converting the comment lines and simply printing the attribute *codeLine* of the generic statement to the output file.

Similar to the new lines, other program elements can be introduced to SCML models as generic statements. In this way, these program elements are preserved; that is they do not get lost during the application of the CDIV. The work-flow steps can also be modeled to add such elements.

Design Idioms and Evolution

We modeled three design idioms, extracted from the latest version of the control software with the domain expert. These idioms are implemented for the interface functions that deal with getting the status of the amplifier. The design idiom used

Idiom	# of Steps	# Alternative Paths
General Status	7	6
Periodic Check	10	3
Sampling	7	4

Table 6.1: The details of the modeled work-flows for the three design idioms used in the gradient amplifier control software

Idiom	Useable		# of times used	# of Annotations	LOC generated
	Verif.	Code			
General Status	Yes	Yes	7	61	158
Periodic Check	Yes	Yes	9	35	95
Sampling	No	No	1	5	0

Table 6.2: Result of application of the three design idioms to the oldest version of the amplifier control software; the implementation and the verification agree on the applicability of the idioms

for getting the status of the amplifier is described in Section 6.1. The other two interface functions are periodic checking and sampling the status registers. Some of the status values in the status registers can only be obtained by sampling. The control software implements a mechanism that reads these values during the scan at very short intervals. When the amplifier is idle, for diagnosis purposes the MRI computer periodically (i.e. at certain intervals) checks the values stored in some status registers (including the ones that are used for the general status). The control software implements mechanisms that read the values of these registers at intervals defined for each register.

Table 6.1 provides the number of distinct steps (i.e graph transformation rules) and the number of alternative paths these work-flows contain. For example, in the work-flow general status certain steps are taken once (depending on the requirements of the register) and because of this, there are six possible ways of implementing this work-flow. The path taken depends on the supplied annotations and on the supplied source code.

Table 6.2 shows the results of the verification applied to the oldest version of the amplifier. This amplifier has 7 general status registers; as a result the design idiom *General Status* is used 7 times. In order to verify these 7 instances of the design idiom, 61 annotations are entered. The verification showed that all 7 instances of the design idiom can be applied without problems. Then, we generated the source

code of the invariants and compared it with the actual implementation. The applied work-flow steps and the actual implementation agreed.

Some invariant statements added by the transformation rules have preconditions that are macro declarations. For example, line 4 of the code segment presented in Figure 6.1 requires the macro *FAN_REGISTER* to be declared (in the driver code). Because the graph-based model allowed such macros to be modeled, the transformation rules were able to verify their existence. To further test this, we removed the macro declaration *FAN_REGISTER*; the verification then failed in applying the transformation rule *initStatusStructs*.

The design idiom *periodic check* is used 9 times; each time the verification and the actual implementation agreed. The verification showed that the design idiom *sampling* could not be used for this amplifier. The oldest amplifier does not allow more than one register to be sampled at a time and the design idiom for the newest amplifier allows this. Although a call-back mechanism is used, the implementation of sampling in the old amplifier is very different from the newest one; that is a new design idiom is used to implement sampling in the oldest amplifier. To verify that the design idiom *sampling* cannot be used for the oldest amplifier, only 5 annotations are supplied. Manual verification requires one to investigate 23 lines of code.

The lines of code generated after applying the approach show that invariants indeed are important in implementing a design idiom. Every statement in the generated code is needed to implement the idiom correctly. From this, we conclude that design idioms are used for evolving the software and our approach is beneficial for verifying the applicability of a design idiom.

6.5.3 ECORE to GXL converter

The details on the design of the ECORE to GXL converter tool are presented in Section 5.3.1. However, to remind the readers about how the tool works we present the sequence diagram showing an example conversion scenario also in this chapter.

Figure 6.9 presents a conversion scenario with two converters. Here, the class *ConcreteConverter* is given an ECORE element to convert. This element has a child element and to convert this element the converter should be identified. The class ConcreteConverter calls the method *Converters. getConverter* to get the converter. The class *Converters* has a list of converters; the method *getConverter* identifies the converter for an ECORE element by iterating through this list and calling the method *canConvert* of each converter. In this scenario the class *ConcreteConverter2* can convert the element. Thus, the method *Converters.getConverter* returns an in-

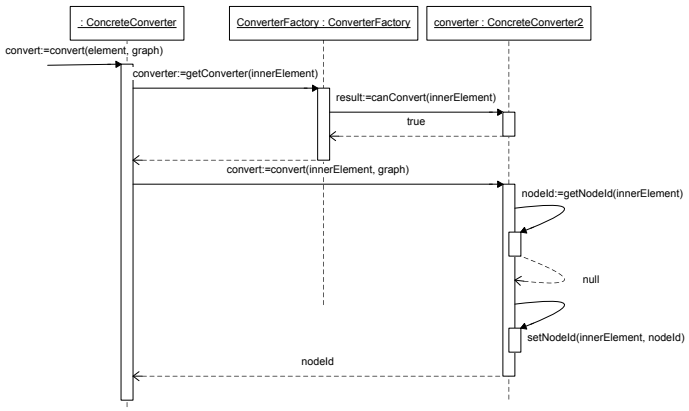


Figure 6.9: An conversion scenario with two converters

stance of this class. The class *ConcreteConverter* calls the method *convert* and passes the child element to start the conversion of this element.

The problem of the ECORE to GXL converter tool is that the meta-model evolves/ changes frequently. When the meta-model evolves, the designer has to quickly evolve the implementation. During this evolution, the designer has to follow the meta-model specification; otherwise, the converter tool would crash or convert with errors.

One frequent type of meta-model evolution is the addition of new classes to the meta-model; we refers to classes in the meta-model as ECORE classes. When a new ECORE class is added to the meta-model, the developers follow the work-flow described below:

1 Add a Java class which is a subclass of the class *ConcreteConverter* that has the name *ModelClassNameConverter*. *ModelClassName* is the name of the newly added class to the meta-model. This should verify that the newly added class is defined in the KM3 language.

2 Write the standard JavaDoc for the Java class added in the previous step.

3 Implement the method *canConvert*.

4 To implement the method *convert*:

   1 Add the calls to *getNodeId*.
   2 Add the calls that convert the ECORE class to a graph node.
   3 Add the code that iterates over the structural features of the ECORE class.
   4 If newly added ECORE class has the attribute called *name*, implement the code that converts the name attribute to a graph attribute.

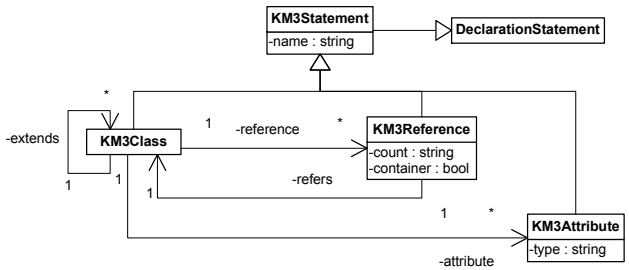


Figure 6.10: The meta-model of elements that is used for representing KM3 with SCML.

5 If the ECORE class has references (i.e children), implement the code that convert each child by calling the method *Converts.getConverter(). convert(). This code should also include calls that add an edge whose name is the name of the reference in the meta-model from the parent object to the child object.*.

The work-flow above is a design idiom imposed by the conversion scenario picked-up by the designer. Every step of this work-flow contains invariants that are crucial for the correct operation of the ECORE to GXL converter tool; for example, if a converter does not implement step 3.1 then the converter would convert the same ECORE object (an instance of an ECORE class) more than once.

The meta-model of the input model to the ECORE to GXL converter tool is specified in the KM3 language and the work-flow steps 1, 4.4 and 4.5 have preconditions that test whether certain things exist in the KM3 specification of the meta-model. For example, the developer has to check whether the ECORE class (for which a new converter is going to be implemented) is specified with KM3 before implementing step 1. These preconditions can be verified with our approach once the SCML meta-model is extended to represent KM3 language. Thus, the implementation of the new converters can be automated with our approach. Moreover, these implementations follow the ECORE class specification without the need for the developer to check anything related to the specified meta-model, which in turn eliminates errors related wrong access of the features of the ECORE class.

In order define work-flow steps with preconditions on the KM3 language, the SCML meta-model should be extended with elements representing the program elements of this language. Figure 6.10 represents the meta-model of the elements used for representing the KM3 language with SCML; note that these elements do not cover all of KM3 language only the elements needed to model the idioms of the ECORE converter tool is modeled. Now that the meta-model is extended, the elements can be used to model transformation rules for work-flow steps 1, 4.4 and 4.5. In the rest of the section the transformation rules of these steps are detailed. However, before

going into the details of these steps, a few points in the control automaton version of this work-flow is explained.

Algorithm 6 The work-flow describing what should be changed/implemented to the ECORE to GXL converter tool when a new model class is added to the meta-model

```
1: function addNewConverter(EClassName){
2: addConverterClass_EClassName;
3: addJavaDoc_EClassName;
4: addMethodCanConvert_EClassName;
5: addMethodConvert_EClassName;
6: implementAddNode_EClassName;
7: implementStructuralFeatureIterator_EClassName;
8: try{
9: implementCopyNameAttribute_EClassName;
10: }
11: while(hasFeatures_EClassName) do{
12: implementConvertChildren_EClassName;
13: implementAddConnectingEdge_EClassName;
14: }
```

Algorithm 6 shows the control automaton, in GROOVE's control language, expressing the work-flow of the steps the developers take to adapt the ECORE to GXL converter tool when a new ECORE class is added to the meta-model. The design idiom with this work-flow is called *addNewConverter* and it takes one parameter named *EClassName* which is the name of the newly added ECORE class to the meta-model. The work-flow consists of 9 graph transformation rules. The text version of the work-flow states that the step 4.4 is a conditional step, in that it is implemented only if the ECORE class has the attribute *name*. This step is modeled with the transformation rule *implementCopyNameAttribute* (detailed in Section 6.5.3), which only matches when the condition on the attribute *name* is satisfied. With the *try* statement at line 9, we instruct GROOVE to go to line 11 if the rule *implementCopyNameAttribute* does not match.

The lines 11 to 13 are used for step 4.5, which states that the conversion code should be added for each reference of the newly added ECORE class. The transformation rule *hasFeatures* match until the conversion code for all references have been implemented; thus, with *while* loop, the transformation rules *implementConvertChildren* and *implementAddConnectingEdge* are repeated until transformation rule *hasFeatures* does not have match.

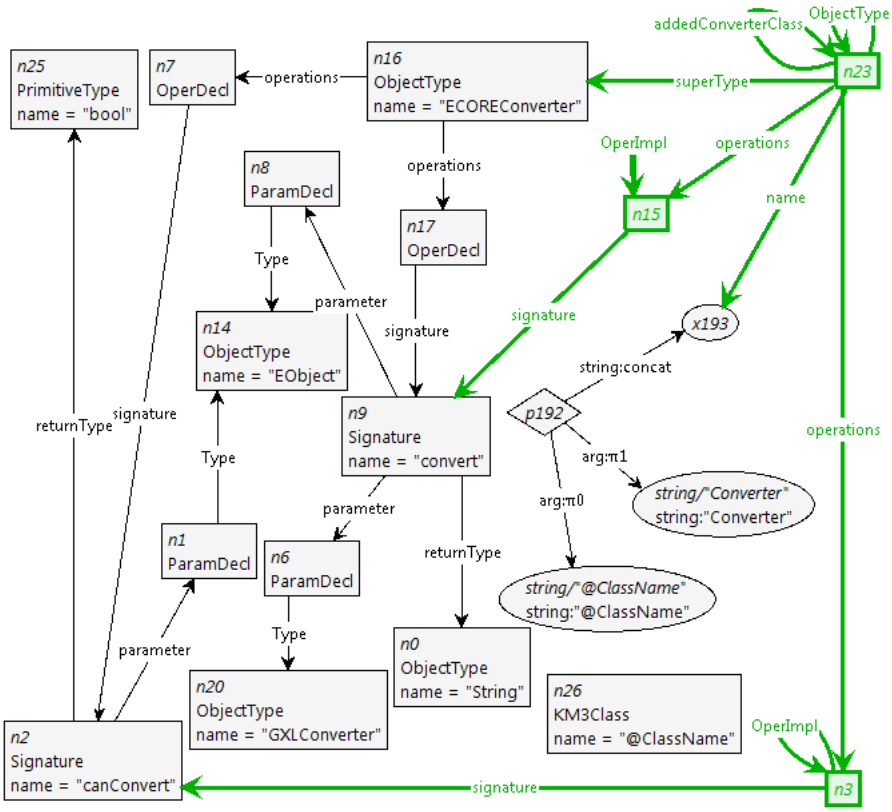


Figure 6.11: The graph transformation rule used for adding a converter class

Adding the Java Class for the New Model Element

Figure 6.11 shows the graph transformation rule *addConverterClass*; the first step of the work-flow. The preconditions of this step are the Java class *ECOREConverter* (each Java class of a converter extends this abstract class), the abstract methods *canConvert* and *convert* and their signatures. In the transformation rule, the node $n16$ represents the class *ECOREConverter*, the operation declaration nodes $n7$ and $n17$ represent two abstract methods belonging to this class and the signatures of these methods are $n2$ and $n9$. All these nodes belong to the left-hand side of the transformation rule, so they should be in the model representing the source code of the ECORE to GXL converter tool for this rule to match. Besides these Java elements, another precondition of this step is the declaration of the ECORE class in KM3 language. In Figure 6.10, we have shown the meta-model of the elements used for representing KM3 language elements with SCML. We choose to use nodes labeled *KM3Class* to represent ECORE classes declared using KM3 language. In the transformation rule, the node $n26$ is an ECORE class declared whose name is

@ClassName. The name *@ClassName* is the parameter of this work-flow step; thus, when the rule is bound (i.e. by replacing the parameter @ClassName with a user specified name) this node would represent the newly added ECORE class. This node is in the left-hand side of the transformation rule to ensure that the rule matches only when the ECORE class is declared.

All the green nodes and edges are added by the transformation rule, which are an object type node (node $n23$) and two operation implementation nodes (nodes $n15$ and $n3$). The Java class added by node $n16$ is connected to the node representing the class *ECOREConverter* with an edge labeled *superType*, this states that the added class is a subclass of the class *ECOREConverter*. Note that the operation implementation nodes are connected to the same signature nodes as the operation declaration nodes (nodes $n17$ and $n7$), this states that these operation implementation nodes implement the methods *canConvert* and *convert*.

The work-flow step also specifies a constraint on the name of the Java class where the converter is implemented; it's name must be the name of the added model element followed by *Converter* The transformation rule achieves this by concatenating the strings *@ClassName* and *Converter*. The node $p192$ is a production node that takes these strings as its arguments. The operator of this production is string concatenation and it is modeled with the outgoing edge labeled *concat*. The concatenated string is assigned to node *x193*, which is the attribute node of the add class. Because the name of added class is computed by the transformation rule, the transformation rule marks the class it adds with the edge labeled *addConverterClass*. In this way, the transformation rules modeling the remaining steps can recognize new Java class.

As discussed before the name *@ClassName* is the parameter of this transformation rule. To add a converter for the KM3 class *Condition*, for example, the user has to specify the annotation *@ClassName = Condition*. CDE tool replaces the string *@ClassName* with *Condition* which binds the transformation rule. In this way, the transformation rule adds a Java class named *ConditionConverter* if all the preconditions are satisfied.

With this transformation rule, we showed the benefit of using models to represent the source code, where the rule verified the preconditions from two different languages. Obviously the KM3 language elements are not in the compiled Java source code and if we were using the Java AST or working directly on Java source files, we would not be able to express this precondition.

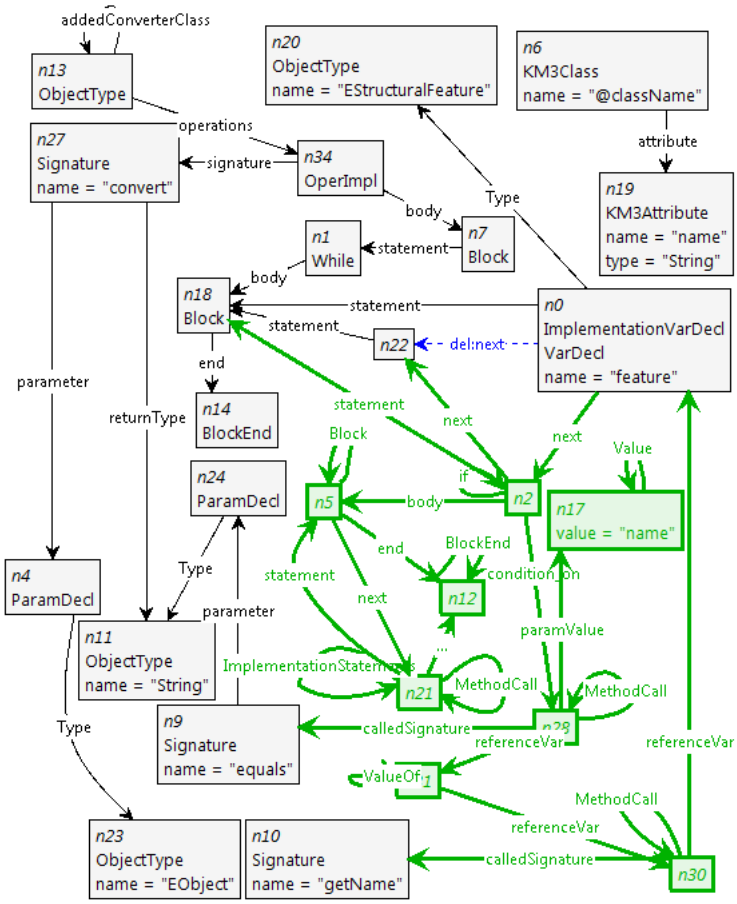


Figure 6.12: The transformation rule that adds statements for converting the attribute *name* to a graph attribute

Converting the Attribute *name*

Similar to the work-flow step detailed in the previous subsection, step 4.4 also has preconditions on the statements from the KM3 language. Specifically, if the ECORE class contains a string attribute called *name*, then two statements are added to the structural features iterator in the method *convert*: an *if* statement for testing whether the iterated structural features name is *name*, and an call statement to the method that converts the attribute of the ECORE class to a graph attribute. These statements form the invariants of this work-flow step. The preconditions are the Java class responsible for converting the ECORE class, the method called *convert* of this Java class, the structural iterator, which is a *while* loop, implemented in the method *convert*, the methods and classes for identifying the name of a structural iterator and the ECORE class with the name attribute.

In Figure 6.12 the transformation rule modeling this work-flow step is presented. Here, the node $n13$ represents Java class added by the work-flow step 1 and it's method *convert* is represented with node $n34$. The structural iterator's while loop is the node $n1$ which is a statement in the method *convert*. Because these program elements are preconditions, the nodes representing these elements are in the left-hand side of the transformation rule. The body of the while loop has an implementation statement is connected to the statement in which the variable *feature* is declared (node $n0$), which models that the variable declaration statement is within the block of statements executed when the condition in the loop is true. This variable declaration statement is also in the left-hand side of the transformation rule because it is a precondition required by the invariants of this step (i.e. the statements added by this step).

The precondition on the attribute *name* is represented with the node $n19$ labeled *KM3Attribute*. This node is the attribute of the KM3 class represented with node $n6$. The nodes and the edge labeled *reference* connecting them are in the left-hand side of the rule; thus, the rule matches when the newly added ECORE class has an attribute called *name* (the rule also checks that the type of this attribute is a string). Other nodes that in left-hand side of the rule are program elements needed by the calls to identify the name of the structural feature, such as the signature *getName* and the class *EStructuralFeature*.

This transformation rule adds an *if* statement represented with node $n2$ with the block of statements that are executed when the condition of this *if* statement is true. Here, the nodes *n5* and *n12* are the block begin and end statements of the added block; only the first implementation statement of this block is shown (node $n21$) the rest is omitted in order to clear the understanding. This *if* statement is inserted after the variable declaration statement node $n0$ and before a statement node 22. Note that the transformation rule does not specify the kind of this statement node (e.g. a the subclass of the implementation statement), so that regardless of the kind of this statement, the rule matches (provided that other preconditions hold).

The condition of the *if* statement is a method call, node $n28$, to the signature *equals*. The value *name*, node $n17$, is passed to the called method by this method call. The reference variable of this call is the value returned by another method call statement, node $n30$, which is a call to the signature *getName*. The reference variable of the second call statement is the variable *feature* (node $n0$). Putting all these nodes together, the condition of the if statement is the statement *feature.getName().equals("name")*, which compares whether the name of the currently iterated structural feature is *name*. These method call nodes and their respective edges are shown in green, so that when the rule is applied an if statement whose condition is these method calls are added.

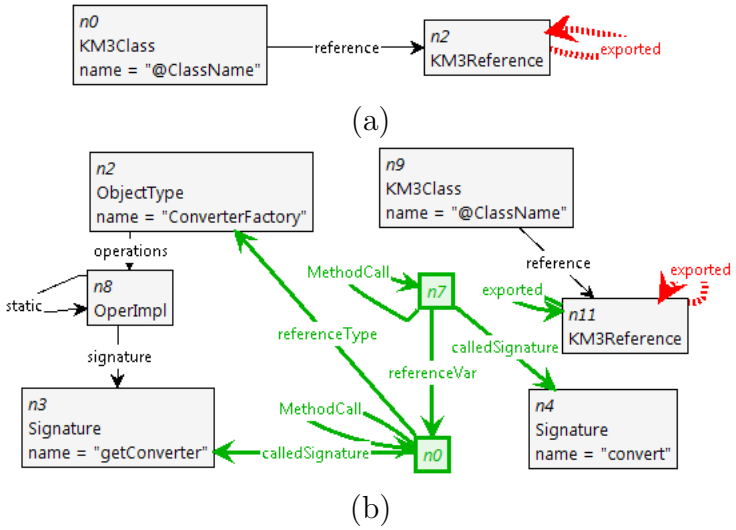


Figure 6.13: (a) the transformation rule *hasFeatures* (b) parts form the transformation rule *implementAddConnectingEdge*.

Converting the Children of the New Model Element

The work-flow step 4.5 is repeated for each reference the model class has. In the control automaton version of the work-flow, we need to have a loop which should terminate when the newly added model class has no reference left for which the statements used for converting references is not added to the method *convert*.

In the control automaton version of the work-flow, the loop is implemented with a *while* statement at line 11 in Algorithm 6. The condition of the while statement is the graph transformation rule called *hasFeatures*; this rule is presented in Figure 6.13-(a). As can be seen from the figure, the transformation rule does not do any modifications to the model of the source. This rule matches when the newly added ECORE class (node $n0$) has at least one reference (node $n2$). The self-edge labeled *exported* on the node representing the reference is a negative application condition.

Assume that, there is an ECORE class with two references called *cond1* and *cond2* and the transformation rule *hasFeatures* matches. When the condition of the while loop is true, then GROOVE tries to apply the transformation rule *implementConvertChildren* and after it, GROOVE tries to apply the transformation rule *implementAddConnectingEdge* (lines 12 and 13 of Algorithm 2). Parts of the transformation rule *implementConvertChildren* is presented in Figure 6.13. This rule picks one of the references of the KM3 class (node $n9$) and marks it with the self-edge labeled *exported*. In this way, the transformation rule *implementAddConnectingEdge* knows

for which reference the statements are added. Now assume that the reference called *cond1* is picked. When the automaton moves back to the while loop after the first iteration, the transformation rule *hasFeature* cannot match to the reference called *cond1*. This is because the edge labeled *export* added by the rule *implementConvertChildren* does not satisfy the negative application condition of the transformation rule *hasFeature*. However, the rule still has a match on the reference called *cond2*. At the second iteration, the self-edge labeled *export* is also added to the node representing this reference. So the transformation rule *hasFeature* will not match and the loop will terminate.

These transformations illustrate how transformation rules can pass information to each other. Here, the edge labeled *export* is used to save the state of the work-flow and it is used by different transformation rules. Because the edge *export* is not in the SCML meta-model, it is ignored by the model-to-source converter.

6.5.4 Verification of the Work-Flow

Due to extensions of the meta-model, 11 more converter classes are added at later versions of the converter tool. Looking at the implementation of these classes, we observed that they all follow the work-flow presented in this section. We initialized the work-flow to add these converter classes using our approach (with 45 annotations). The verification showed that all these classes can be added using the work-flow. Thus, the verification and the actual implementation agreed. Because the rules are modeled in the domain of SCML, adding JavaDoc comments and verifying preconditions on both KM3 and Java statements is easily handled in the process. This shows that the transformation rules can check any precondition in the domain of the graph-based model, regardless of the language the source file is programmed in.

6.6 Related Work

Program transformations are automated manipulations of programs where the program itself becomes the data and the transformation rule provides a high level language for the manipulation [122]. The proposed approaches for program transformations provide source-to-source program transformations, where the source code is transformed into abstract syntax tree (AST) and then the transformations are applied. The transformed source code is then recompiled into source-code format [18, 115, 73]. This complicates rule design. As a result, approaches that allow the rule designer to write rules embedding fragments from the target language [121, 54] and that allow the rules to fully written in the target language [120]

are proposed. From the transformation approaches, Stratego [121] provides a programmable strategy for rule application. Because, all these approaches apply the transformations on the ASTs, the source-to-AST and AST-to-source transformation looses information such as references to macros and comments. The AST is at a lower level then the source code the developers work on, this makes it hard for the developers to express rules. For example, in the control software the developers use refer to register addresses with the macros defined in the driver. The developers do not know the values these macros replace; so the developers would not be able to express a rule at the AST level. Moreover, the AST-to-source code conversion looses comments. In this chapter, we provided an example of a comment line that plays a crucial rule for maintainability purposes. Obviously, a source code without such comments would be hard to maintain. In our approach, SCML supports macros and comments so the source-to-SCML and SCML-to-source conversion does not loose these statements. The rules are specified in the domain of this graph-based model and the transformations are applied in this domain. The graph-based model is a representation of the program as seen by the developer; thus, the transformations do not require any reference to the AST (e.g. the relative location to the AST). If, for example, the source in consideration contains statements that is not covered by the SCML meta-model (i.e. statements from a domain specific language), these statements can still be represented in generated models as generic statements (with the attribute *codeLine* is set to the actual statement). In this way, CDIV can still be used to implement the design idiom and these statements would be preserved in the generated template source code. The applications of the approach showed example, in which, the transformation rules added comments or checked preconditions on the macro definitions. We also showed an example where the SCML meta-model is extended to verify the existence of statements from other languages then the host language the design idiom is implemented in.

In literature refactoring transformations are proposed to improve the quality of the source code [56]. The proposed approaches for automating refactoring transformations do not provide formal models for specifying and verifying the work-flow, the refactoring transformations and they are language dependent [105], [117]. For example, work-flows that have conditions or alternative paths, cannot be specified with these approaches. The approaches that formalize the refactoring transformations, on the other hand, do not provide a way to combine these transformations in a work-flow [99]. The industrial case study presented uses a work-flow that has conditions and we showed that how such complex work-flows can be handled with our approach.

Whittle et. al [126] uses the transformations to automate certain operations in UML. In this approach, the transformations are defined in the context of UML class diagrams that are annotated with OCL constraints. The transformation rules

are used for refactoring/evolving model elements by obeying the OCL constraints. SCML can easily accommodate UML class/sequence diagrams. Thus, with our approach it is also possible to define transformations on UML diagrams or define rules that co-evolve UML and source-code. We are investigating the possibility of the combining the approach presented in Chapter 5, to enforce constraints while evolving the model.

Mens et. al [96] use declarative programming to generate code and detecting violations on best practice patterns. The declarative rules describe how the pattern is applied and the pre-conditions of the pattern. With Prolog [38] like evaluation of the declarative rules the pattern is applied. Here, the application order is dependent the derivation of the rules; that is, the user has no control over the derivation order of the rule. Thus, a work-flow specification is not possible with this approach. Moreover, with this approach patterns that have alternative patterns cannot be handled when both alternative evaluate to true. In our case, the user can generate code for each alternative by clicking on an alternative path in work-flow. We also provide a visual representation of the work-flow, so the user can manually undo, redo steps or generate code up to a certain step in the work-flow.

JML [87] is a behavioral specification language for Java. The java source code is annotated with JML specifications describing the program invariants. A sub-set of this specification language is used to ESC/Java2 [37] provided static checking of invariants. This verifier is not suitable for the case studies presented in this chapter. Because the annotations aim for correct usage of the annotated code (e.g. methods), where as, in the case studies the developer implements a new software artifact (e.g. a class) by following an idiom (or a pattern). In a sense, the developer implements the code to be annotated. Our approach helps the developer by automating the verification of the pre-conditions of the idiom. If the verification succeeds our approach provides template source code to provide guidelines one on the implementation of the design idiom.

Design patterns can be used to ease certain evolutions. However, due to poor documentations their effectiveness is reduced. In our previous study [32], we used graph transformations to correctly evolve design patterns. Kobayasahi et al. [84] define steps and languages for instantiating and evolving a template design pattern. Zhao et al. [133] present how design patterns can be evolved correctly using graph transformations. These approaches all lack a work-flow specification. Thus, they cannot be used work-flow that have loops and conditions.

6.7 Conclusions and Future work

This chapter presented a process and tools for computer aided verification of the usability of a design idiom. In this process the work-flow of the design idiom is modeled. Each step of this work-flow models the invariants that have to be implemented at that step. The invariants are statements that are imposed by the design idiom and are crucial for correctly implementing the design idiom. These invariants have preconditions; that is, they require/depend on other software entities. To test the applicability of a design idiom, the tool tries to simulate the work-flow. In each step of the work-flow, the tool checks if all the preconditions are met. If a precondition check fails, the design idiom cannot be applied and the tool presents the failed step to the developer.

The approach uses an extensible meta-model for representing and transforming the source code called SCML. SCML includes program elements like comments and macros. As a result, the design idioms can be expressed and checked at level of the source code the developer works with. The code generated after the use of the approach also includes and preserves these elements.

The models in SCML are attributed graphs. The semantics of the steps are modeled using graph transformation rules. The left-hand side of the rule includes the preconditions and the right-hand side adds the invariant statements to the model of the source code. The work-flow of the design idiom is modeled using state machines. The work-flow and the graph transformation rules are stored in a repository.

In different implementations of a design idiom, the names of the program elements like variables may change. In order to compensate such changes the names of the software entities are parameterized. To use a design idiom, the developer specifies the name of the idiom and the source files needed to implement this idiom. The source files are converted to models in SCML. The parameters of the transformation rules the idiom uses are automatically extracted and presented to the developer. The developer provides the actual names (names used in the source code) of these parameters. Using the actual names, the graph transformation rules are bound. A modified version of GROOVE is launched, which applies the transformation rules by following the work-flow of the idiom. After the application a verification algorithm checks if a path from the start state to a final state in the generated state-space is also a path from a start state to a final state.If it is, then template source code that correctly implements the invariants of the design idiom is generated. If the verification fails, then feedback about the failing step is presented to the developer.

SCML is designed to represent source code the developer see. In the current embedded systems, the use of domain specific languages embedded in a host language like

C is extensively used. Because SCML treats every program element as a statement the meta-model extension to cover these language can be implemented. However, such implementations require changes to source-to-SCML and SCML-to-source converters. Although the tool set is designed to be extensible, such an extension still requires the conversion algorithm to be implemented. With general purpose test-to-model converters meta-model extension can easily be implemented (i.e. no need to implement a conversion algorithm). Using a general purpose text-to-model converter, we implemented a proof-of-concept Java-to-SCML converter. We identified certain points that needs to be improved in these general purpose converters while implementing this proof-of-concept converter. As a future work, we are going to implement these improvements. Thus, the extensibility of SCML can be better put into practice.

Chapter 7

Conclusions

7.1 Problems Addressed in this Thesis

Evolving and maintaining software systems is an error-prone and time-consuming task. The major cause of these errors and time is the manual testing on whether the software can reach a desired evolved state by some evolution procedure. This thesis addresses the following instances of the manual testing problem:

- **Runtime Reconfiguration:** Runtime reconfiguration is used heavily in today's software systems in order to adapted the software systems to the user's needs without recompilation and on site. How the software system reconfigures itself is usually described in the design models. However, due to the lack of tools that allow testing of the reconfigurability of these models, the testing is postponed until after the software is implemented. This in turn increases testing time as more test cases needs to developed, and the fixing of design errors in the implementation requires more effort.

- **Design Constraints:** The developers tend to reuse software designs for requirements that are similar to previous requirements. These designs usually have many constraints that are crucial for the correct operation of the software and in each implementation of the design the developers have to make sure that they satisfy the constraints of the design. In addition to these constraints, the developers are also required to satisfy coding conventions that are enforced by the organization. Keeping up with these coding conventions and design constraints is an effort and time consuming task.

- **Design Idioms:** Sometimes the reuse of the design involves adapting/changing

one of the previous implementations of the design to address new requirement. However, the developers have no way of knowing whether this adaptation violates the invariants of the design, which means that they cannot use the design for the requirement. Due to the lack of such knowledge, bugs may be introduced to the software. Even worse, the developers may need to re-implement the requirement with another design.

7.2 Solution Approach

In this thesis, we proposed the evolution simulation approach. The type of the evolution or, in other words, how the software system is evolved is modeled; these models are called change operators. To evolve the software, the developer enters the desired evolution operators and the source code, the accompanying tools of the approach tries to evolve the software using the supplied operations. The evolution simulation approach is specialized to address the three problems described in the previous section. Below these specializations are summarized:

- **Computer-aided runtime reconfiguration requirements verification of UML models:** A software system may support many reconfigurations and may consists many sequence diagrams, thus, manually tracing these diagrams for reconfiguration requirement verification may consume too much effort. We address this problem by providing an environment in which the sequence diagrams are simulated. Because runtime reconfiguration happens while the system is operational, the effects of it are best captured when the simulation is close to the actual execution environment of the software system. We modeled execution and reconfiguration semantics for UML sequence diagrams that are close to the actual execution of the object-oriented systems. The simulation of the UML models with these semantics generates all possible execution sequences the models support; each execution sequence, here, is a configuration.

  Using computational tree logic or a visual state based language the reconfiguration requirements are expressed as execution sequences. A verification algorithm searches whether there are execution sequences generated by the simulation which matches the execution sequence expressed in the reconfiguration requirement.

- **Graph-based static design constraint checking:** We developed a modeling language called Source Code Modeling Language (SCML) for representing the source code and expressing the constraints with the program elements

that are at the source code. The source code to be verified with the design constraints is converted into an SCML model, which is an attributed graph. The pattern of the constraint is expressed using graph transformations, where, the right hand-side is used for extracting information for error reporting. A transformation engine tries to apply the transformation rules modeling the constraints. If a rule matches, then the constraint modeled by the rule is violated so the right hand-side of the rule adds a node whose attributes describe the violated constraint, and the physical line number of the statement(s) that violate the constraint.

SCML model of source code may contain many elements, so it may be hard to locate the elements about the violated constraints added by the transformation rules. We have built a querying mechanism based on predicate logic that retrieves this information from the graph.

- **Computer-aided design idiom verification:** The design idioms consist of a work-flow and the steps of the work-flow in which the invariants of the idiom are implemented. We used a control automaton for modeling the work-flows. The steps of the work-flow are modeled as graph transformation rules in SCML; thus, the work-flow steps include all the program elements visible at the source code. Here, the checking involves whether the requirements of the invariants are satisfied are not. A requirement, for example, can be a constraint on the type of a variable.

  The transformation rules are modeled as templates that is they parameterize the names of the program elements. To check whether a design idiom can be used, the developer supplies the source code and the name of the design idiom. The tools convert the source code into an SCML model, gather the transformation rules used in the design idiom and form a binding file that contains all the parameters. The developer fills this binding file and submits it to the tools. Using the binding file, the parameters are replaced with the actual names of the program elements. The bound transformation rules and the work-flow is loaded to a graph transformation tool. The transformation rule follows the work-flow while applying the transformation rules. If a work-flow step cannot be applied, then the requirements of the invariants are missing. The transformation tool reports the failed step to the user.

7.3 Contributions

Below the contributions according to the addressed problems of this thesis are listed.

Computer-aided runtime reconfiguration requirements verification:

1. The Design Configuration Modelling language which is an object-oriented runtime representation for UML class and sequence diagrams. This model includes means to incorporate the reconfiguration mechanisms the design uses.
2. Graph transformation rules modeling the execution and reconfiguration semantics for simulating the DCML models, which in turn allows the simulation of sequence diagrams. These semantics are close the actual object-oriented system execution.
3. A method for expressing the reconfiguration requirements as CTL formulas and verifying these requirements of the all execution sequences the UML diagrams support.
4. Visual State Based Language (VSL), for expressing the reconfiguration requirements visually.
5. Methods for getting feedback on the reconfiguration errors based on CTL and control automaton

Computer-aided static design constraint checking:

1. A meta-model for representing the program elements at the source code level and expressing constraints over these elements.
2. A process for checking design constraints at the source code level.
3. A detailed error reporting mechanism to provide better guidelines to the developers.
4. A method for using graph-transformations and graphs to visually express constraints over complex patterns.

Computer-aided design idiom verification:

1. The use of meta-modeling that allows design idiom verification/implementation at the source code level.
2. A method for modeling the work-flow of the design idioms using control automaton.
3. A method for visually modeling the steps of the work-flow using graph transformations
4. A process for simulation of design idiom implementation and verification of whether the work-flow is completed or not.

7.4 Future Research Directions

This thesis explained various applications of graph-based model checking to specification software evolution problems. These applications lead to further interesting research directions that we will investigate in the future. Below we provide these two directions:

Preventing reconfiguration errors on the actual running system: Due to the lack of values in the UML modeling phase, the approach we propose to early evaluation of runtime reconfiguration in Chapter 3 does not help in verifying the configuration system once the OO-design is implemented. Even though, it is possible to achieve correct interactions w.r.t reconfiguration using our approach, errors may still be introduced when the design is implemented. Especially, the configuration system may need to handle many values and send them to the application. It is possible to use an OO-program level model checking approach to verify that the values are also correct; however, with these approaches the generated state-space is usually very large hampering the scalability. To address the scalability problem and to verify the values are also correct, we will focus on a runtime observation mechanism using aspects. This mechanism will observe the current runtime and compare it with the generate state-space. Since the state-space contains what a value should be or not be for correct reconfiguration, the observer can recognize when the software is doing something is should not do. The observer can then trigger a recovery system to put the software back into a correct state. Here, the research problems is on providing a dynamically adjustable point cut system and the local recovery techniques.

Source-to-Model conversion using higher order transformations: The discussion on chapters 5 and 6 aims at providing a modeling language called SCML for source codes where different languages can be represented with the program entities. For example, with SCML one can easily co-evolve to languages such as Java and C. However, one still needs to program a source-to-SCML converter to introduce a new language to be represented in SCML. Since the SCML meta-model is known, it may be possible to generate the converter once the grammar of the language is specified. Using the grammar and a parser generator like ANTLR [1] the parser for the language can be generated. This parser outputs the concrete syntax tree of the program. After this, to generate the SCML model one needs to program the converter, which can be a very complicated task. We plan to address this problem by using higher order transformation rules that will parameterize the recognized concrete syntax elements. The user will only provide the mapping form the

concrete syntax elements to the SCML elements. The higher order rules will use the mapping and correctly generate the SCML model for the new language. The research questions involving this is the use of higher order rules and how to parameterize the recognized entities.

Appendix A

UML-to-DCML conversion in detail

We implemented a UML-to-DCML converter that is built into ArgoUML [2] using ArgoUML XMI API. The choice for using ArgoUML is that it supports the two mostly used XMI manipulation libraries, MDR and eUML, and provides an open source library that hides the details of the underlying XMI library. In addition to these, ArgoUML conforms with the XMI specification, so it can be used to load UML models drawn using other tools. One can use ArgoUML to model the software or load the XMI file of the UML models drawn with another tool and use ArgoUML to only convert the UML models to DCML models.

A successful conversion generates a GXL file which contains the DCM in graph form. It is important to note here that the graph generated from the translator is an edge labeled directed graph, which is the type of the graph GROOVE supports. The nodes are distinguished with unique identifiers in this type of graph; the unique id is stored in the GXL file and it is of the form nX where X is an integer number. A node's label is shown with a self edge with the same label. An edge in the GXL file is identified with the identifier of the nodes the edge is connecting.

The UML-to-DCML converter executes in two steps. The first step converts the class diagram and the second step converts the input sequence diagrams. Each step consists of two parts: a model traverser and converters. A model traverser is simply a tree traversal algorithm that traverses through the elements of the input UML models. The converters are a set of classes that implement a conversion algorithm for various UML elements. This section details the conversion process for both steps of the converter.

A.1 Class Diagram Conversion

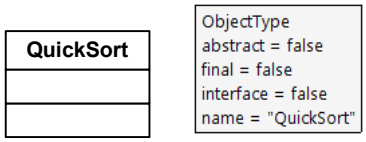
The class diagram conversion is realized by traversing through all the elements of the class diagram and calling the appropriate conversion algorithm for the traversed element. We programmed a specific converter for each type of element in the class diagram; for example, a converter only handles the conversion of UML classes. The elements in UML are identified by their names whereas in the GXL form the edges refer to nodes with their identifiers. Because of this, the UML-to-DCML translator uses a *symbol-table* to resolve the node identifiers during conversion. The symbol-table is a hashtable indexed by the name of the UML elements and the cells contain the unique identifiers. For an element in the class diagram, the class diagram conversion first tries to resolve the node identifier from the symbol-table. If the node identifier is not located in the symbol-table then the translator executes the conversion algorithm for that UML element type. The name of the UML elements are indexed according to the following format: $<containerName>elementName$. Here, the *containerName* is optional and it refers to the name of the class or the interface to which the UML element belongs. For example, the node identifier of the attribute a of class foo is resolved by the key $fooa$. The optional key *containerName* is not used for elements of type class, interface and type.

Algorithm 7 presents pseudo-code for the methods used for converting UML classes and UML class attributes to clarify how the symbol-table and the traversal of UML class diagrams work. Here, the algorithm that converts the UML classes is presented by the method *convertClass*. The UML class elements are converted to object-type nodes with the same name. Before creating a new object-type, the algorithm first checks if an object-type node already exists that represents the class that is being converted. It is an important step, taken to prevent creating duplicate DCML elements for the same UML class. If the symbol-table does not contain an object-type node representing the class, the *if* statement at line 2 evaluates to *true*. At this step, the algorithm adds an object-type node (line 4) and, then, adds an entry to the symbol-table for the newly added node (line 5). After these steps, the conversion algorithm converts the attributes and the operations of the class as shown in lines 6-15.

The method *convertAttribute()* presents the pseudo-code of the algorithm used for converting the attributes of the classes. This method starts by checking whether the attribute is converted before; if it is converted before, then the node identifier is retrieved from the symbol-table. If, on the other hand, the attribute is not converted before, then the method adds a variable declaration representing the attribute (i.e. it has the same name as the attribute) at line 23. The entry for this attribute is added to the symbol-table with the method call shown in line 24. After adding

Algorithm 7 The pseudo-code for methods used for converting UML classes and class attributes to DCML.

```
1: method Integer convertClass(UMLElement e){
2: Integer id = symbolTable.getId(e.getName());
3: if id = NULL then
4:     id ← addObjectTypeNode(e);
5:     symbolTable.addEntry(e.getName(), id);
6: end if
7: for s ∈ e.getChildren() do
8:     Integer childId;
9:     if s.type() = "attribute" then
10:        childId ← convertAttribute(s, e.getName());
11:        addEdge(id, childId, "attributes");
12:    end if
13:    if s.type() = "operation" then
14:        childId ← convertOperation(s);
15:        addEdge(id, childId, "operations");
16:    end if
17: end for
18: return id;
19: }
20: method Integer convertAttribute(UMLElement attribute, String containerName){
21: Integer id ← symbolTable.getId(containerName + e.getName());
22: if id = NULL then
23:    id ← addVariableDeclarationNode(attribute.getName());
24:    symbolTable.addEntry(containerName + e.getName(), id);
25:    Integer typeNodeId ← symbolTable.getId(attribute.getTypeName());
26:    if typeNodeId = NULL then
27:        typeNodeId ← addTypeNode(attribute.getTypeName());
28:    end if
29:    addEdge(id, typeNodeId, "Type");
30: end if
31: return id;
32: }
```



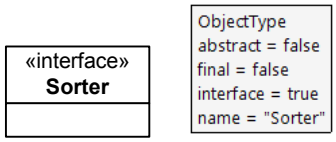
(a) (b)

Figure A.1: a) UML Class of the class *QuickSort* b) the UML Class *QuickSort* in DCML.

the variable declaration node, the method starts the conversion of the type of the attribute. This conversion is done similar to the attribute conversion where the method first tries to locate the node identifier of the type as shown in line 25. If the type is not located in the symbol-table, then the method *addTypeNode()* is called which implements the algorithm used for converting UML type elements. Once the identifier of the node representing the type of the attribute is located (or generated by adding the node), the method adds an edge labeled *type* connecting the variable declaration node representing the attribute to the located type node. The identifier of the added/located variable node is returned to the method *convertClass()* at line 10. The method *convertClass()* then adds an edge from the object-type node to the variable declaration node labeled *attributes*.

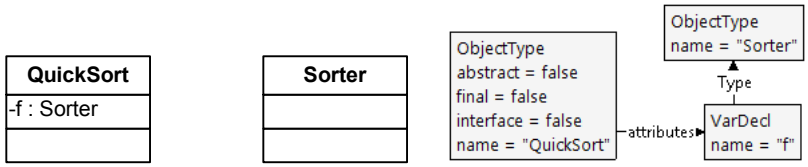
Below, the conversion of UML class diagram elements are detailed with example UML-to-DCML conversions:

- *Type Elements:* If the UML type element is a primitive type then it is converted to a primitive type node. Complex types are converted to object-type nodes. The attribute name of the type node (primitive or object-type) is set to the name of the UML type element.
- *Class Elements:* Class elements are converted to object-type nodes. If the class is an abstract class then the attribute *abstract* of the object-type node is set to *true* and the remaining attributes are set to *false*. Similarly for final classes, the attribute *final* of the object-type node is set to *true* and the remaining attributes are set to *false*. Note that execution semantics discard the attributes of the object-type nodes; they are only present in DCML models to ease the understanding of the relation between DCML and UML. Figure A.1 presents the class *QuickSort* in UML (a) and in DCML (b).
- *Interface Elements:* The UML interface elements are converted to object-type nodes with the attribute *interface* set to *true* as shown below in Figure A.2.
- *Attribute Elements:* Attributes of classes are represented by variable declaration nodes that are connected to the respective class with an edge labeled



(a) (b)

Figure A.2: The interface *Sorter* in UML (a) and in DCML (b)



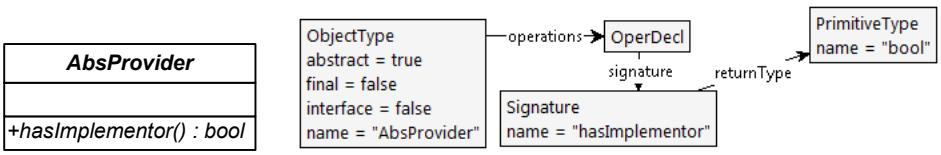
(a) (b)

Figure A.3: The class *QuickSort* with attribute *f* in UML (a) and in DCML (b).

attributes. In Figure A.3-(a) the class *QuickSort* with an attribute named *f* is shown in UML; in Figure A.3-(b), this class is shown in DCML where the attribute *f* is now a variable declaration node

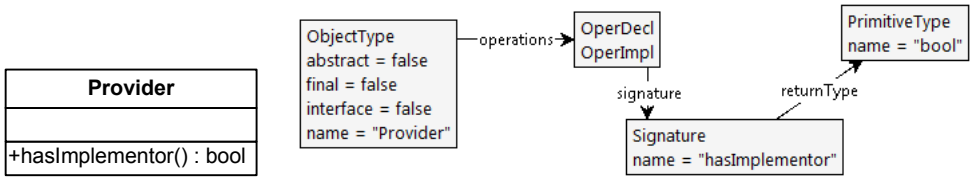
- *Operation Elements, abstract methods of a class:* The abstract methods are converted into operation declaration nodes and signature nodes. It is important to note that in DCML every unique signature is represented by a signature node. Two operation nodes with the same signature are connected to the same signature node even though they belong to different classes. The return type of a method is shown in DCML by connecting the signature node to the type node with an edge labeled *returnType*. Figure A.4-(a) shows the abstract class *AbsProvider* with the abstract method *hasImplementor()* in UML and Figure A.4-(b) shows this abstract class in DCML.

- *Operation Elements, methods of an interface:* Because in interfaces the methods are not implemented, the methods of an interface are represented with



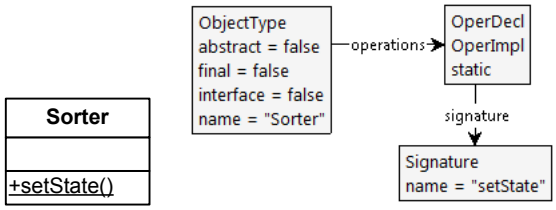
(a) (b)

Figure A.4: The abstract class *AbsProvider* with the abstract method *hasImplementor()* in UML (a) and in DCML (b).



(a) (b)

Figure A.5: The class *Provider* with the implemented method *hasImplementor()* in UML (a) and in DCML (b).



(a) (b)

Figure A.6: The class *Sorter* with the static method *setState()* in UML (a) and in DCML (b).

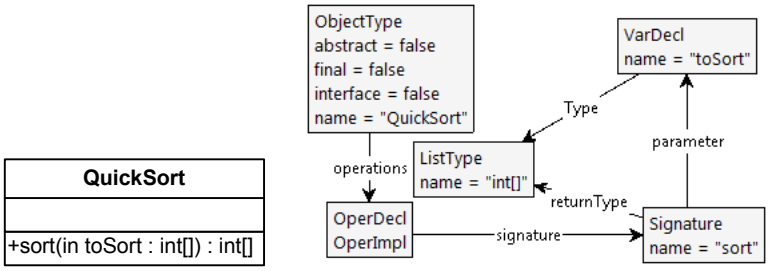
operation declaration nodes connected to the object-type node representing the interface with an edge labeled *operations*.

- *Operation Elements, implemented methods:* The methods that are not abstract are represented by nodes labeled both operation declaration (*OperDecl*) and operation implementation (OperImpl) as shown below in Figure A.5.

- *Operation Elements, static operations:* Static operations are represented by operation implementation nodes (nodes labeled both *OperImpl* and *OperDecl*) that are also labeled static. The class diagram of the class *Sorter* with the static operation *setState()* is presented in Figure A.6-(a). In DCML, this method is represented by the operation implementation whose signature is named *setState* as shown in Figure A.6-(b).

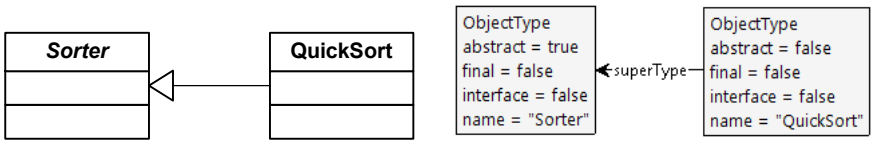
- *Method Parameters:* The method parameters are represented by variable declaration nodes that are connected to the respective signature nodes with edges labeled *parameter*. In Figure A.7-(a), the method *QuickSort.sort()* that takes a parameter named *toSort* is presented. This parameter is represented by the variable declaration node named *toSort* in the DCML model presented in Figure A.7-(b).

- *Generalization of Classes:* The generalization relation between classes is converted to edges labeled *superType*. Figure A.8-(a) shows a UML class diagram



(a) (b)

Figure A.7: The class *QuickSort* with the method *sort()* that takes an integer array in UML (a) and in DCML (b).



(a) (b)

Figure A.8: The class *QuickSort* generalizing the abstract class *Sorter* in UML (a) and in DCML (b).

where the class *QuickSort* generalizes the abstract class *Sorter*. This class diagram in DCML is presented in Figure A.8-(b). In this DCML model, the object-type node labeled *Sorter* represents the class *Sorter* and the object-type node labeled *QuickSort* represents the class with the same name. Because the class *Sorter*, is abstract the attribute *abstract* is set to true for the object-type node representing this class. The edge labeled *superType* shows that at runtime the object-type *Sorter* is a super-type of the *QuickSort*. This edge represents the generalization relation between the respective UML classes in the UML class diagram.

- *Realization of Interfaces:* Because interfaces are treated as types at runtime, the realizations between an interface and a class are also represented by edges labeled *superType* in DCML.

- *Abstract Method Implementation:* Abstract method implementations are modeled by connecting the operation declaration node representing the abstract method and the method that implements it to the same signature node. In Figure A.9-(a) a UML class diagram showing the implementation of an abstract method is shown. Here, the abstract class *AbsProvider* declares the abstract method *hasImplementor()*. The class *Provider* generalizes the class *AbsProvider* and implements this abstract method. The DCML equivalent

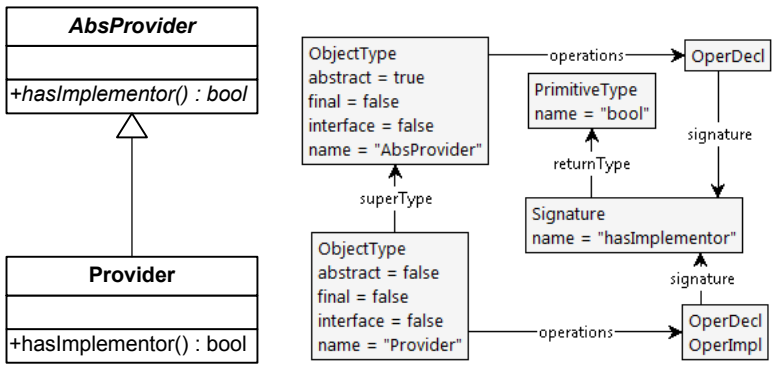


Figure A.9: The class *Provider* implementing the abstract method *hasImplementor* in UML (a) and in DCML (b).

of this class diagram is shown in Figure A.9-(b). The generalization relation between the classes *AbsProvider* and *Provider* is represented with the edge labeled *superType* connecting the object-type nodes *AbsProvider* and *Provider*. Note that there is only one signature node in the DCML model because the class diagram has only one unique signature, which is the signature *hasImplementor*. This signature node is connected to one operation declaration node (node labeled *OperDecl*) and to one operation implementation node (node labeled *OperImpl* and *OperDecl*). The operation declaration node belongs to the object-type *AbsProvider*; thus, this operation declaration node represents the abstract method *AbsProvider.hasImplementators()*. The operation implementation node, on the other hand, represents the method *Provider.hasImplementator()*.

- *Method Overriding:* Method overriding is modeled by connecting the operation implementation nodes of the overridden and the overriding methods to the same signature node. The UML class diagram in Figure A.10-(a) shows the method *Provider2.hasImplementor()*, which overrides the method *Provider.hasImplementor()*. The DCML model of this class diagram is presented in Figure A.10-(b). Here, the object-type node *Provider* is a the supertype of the object-type *Provider2*. The operation implementation nodes belonging to the object-type *Provider2* and to the object-type *Provider* are connected to the signature node named *hasImplementor*. This shows that the operation *Provider2.hasImplementor()* overrides the method *Provider.has -Implementor()*

- *Association between Classes:* DCML only supports interaction between objects through encapsulation. Because of this the associations between classes are

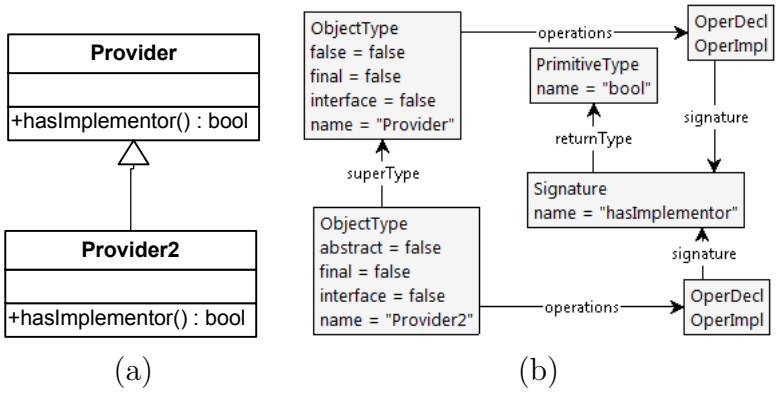


Figure A.10: The class *Provider2* overriding the method *hasImplementor* in UML (a) and in DCML (b)

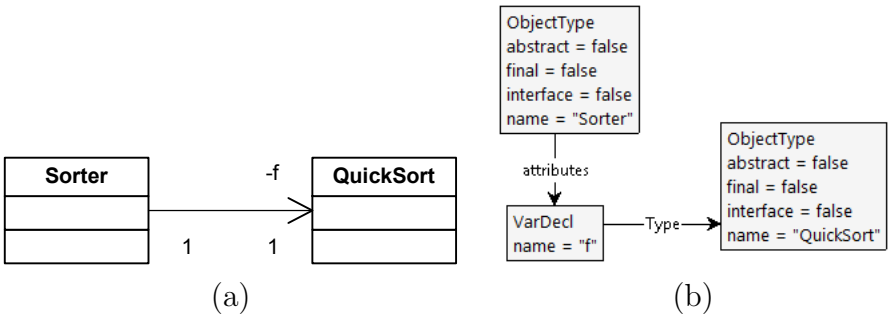


Figure A.11: a) UML class diagram showing the association between classes *Sorter* and *QuickSort*. b) The association shown in (a) is represented as an attribute in the DCML model of this class diagram.

converted to attributes. The *end* name of the association is used as the name of the attribute as shown in Figure A.11.

A.2 Sequence Diagram Conversion

Similar to the conversion of class diagrams, the sequence diagram converter also traverses the actions of the sequence diagram and adds the appropriate DCML elements. The conversion runs the traversal twice: the classifiers are converted to object value nodes in the first traversal and the call/return actions are converted into DCML action nodes in the second traversal. The conversion uses the symbol table generated by the class diagram converter. Besides this table, the converter uses two more tables.

The first table, called *valueObjectTable*, stores the mapping from the owner variable (i.e. the variable that holds the value) to the value/object nodes. This table is filled by the first traversal of the sequence diagram, which only converts the classifiers of the sequence diagram to object/value nodes. This traversal checks if the traversed UML element is a classifier and, if so, it calls the method shown in Algorithm 8. The UML-to-DCML conversion requires all the names of the classifiers to be specified in the form *name1, name2*. From these names, the first name is used as the *owner* variable. When the sequence diagram is converted to a DCM, the names of a classifier are used as the names of the variables the object-types use to access the object or the value. The owner variable, or the first name of the classifier, states that the object/value is first accessed in the scope of the variable with the same name. Thus, the converter connects the object/value node representing the classifier to this variable (in this way passing of values/objects can be simulated accurately). Following this, the method *convertClassifier()* extracts the first name of the classifier as shown in line 2 of Algorithm 8. Then, it adds a value node representing the classifier. If the type of the classifier, accessed through the call *classifier.getBase()* in line 4, is a class then the value node is also labeled *object*. Finally, the algorithm adds the identifier of the object/value node to the table *valueObjectTable* with the key value hold by the array element *names[0]*. This array location holds the first name of the classifier; so, in the *valueObjectTable* the identifiers of the value nodes are indexed according to the name of the owner variable.

Algorithm 8 The pseudo-code for the method used for converting classifiers

```
1: method Integer convertClassifier(UMLElement classifier){
2: String[] names ← classifier.getName().tokenize(",");
3: Integer valueObjectNodeId ← addNode("Value");
4: if !classifier.getBase().isPrimitive() then
5:     addEdge(valueObjectNodeId, valueObjectNodeId, "Object");
6: end if
7: addEdge(symbolTable.getId(names[0], valueObjectNodeId, "instance");
8: valueObjectTable.add(names[0], valueObjectNodeId);
9: return valueObjectNodeId;
10: }
```

The second table, called *acttionTable*, stores the mapping from the call/return actions to the identifiers of the action nodes (call or return). After converting the classifiers, the sequence diagram converter traverses the sequence diagram again to convert the actions. The elements of the UML diagrams are stored in the order they are drawn in the *XMI* files. Because of this, the traversal of the actions may not follow the order in the sequence diagram. For example, if the last action drawn by the user is the first action of the sequence diagram then the last action the converter sees is this first action. The order between the actions is stored in the activator and the predecessors of the actions; the converter follows this order while converting the

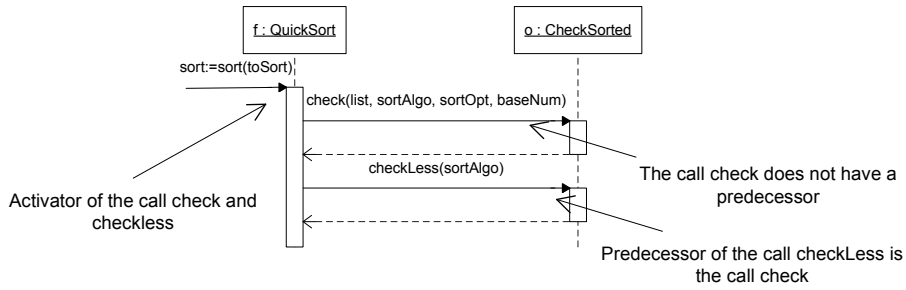


Figure A.12: An example sequence diagram with the activator and predecessor of the call actions detailed.

actions. The activator is the call action after which the object starts executing; that is, the call after which a new activation bar is added to the life-line of the object. The predecessors of actions, on the other hand, are used for ordering the actions within an activation bar. Figure A.12 depicts the relation between the activator and the predecessors. Here, an instance of the class *QuickSort* makes two calls, namely *check* and *checkless*. These calls belong to the activator (or activation bar) that starts with the call *sort*. Thus, the activator of these two calls is the call *sort*. The call action *check* does not have a predecessor because no other call action comes before it. The predecessor of the call *checkLess* is set to the call *check* to specify the order between these actions, which states that the call *check* comes before the call *checkLess*.

After having clarified the definitions of activators and the predecessors of sequence diagram actions, we will explain the method *convertCallAction()* used for converting call actions. The pseudo-code for this method is presented in Algorithm 9 and it is called when sequence diagram traversal encounters a call action. Because out of order call actions can be encountered during traversal, the method first checks whether the call action has been already converted, at lines $2-4$. Then, it checks whether the activator of the call action is already converted. If the activator is not converted, then it makes the calls to convert the activator as shown in lines $5-8$. It is important to note here that the method identifies the first call action of a sequence diagram (i.e. the call action that initiates the sequence) by checking whether the activator is send from an anonymous classifier; an anonymous classifier is a classifier with no name and type (or base).

After the checks for the activator, the method forms a string that uniquely identifies the operation implementation node that was called by the activator. As discussed in the previous section, during class diagram conversion a symbol-table is formed containing the name to node identifier mapping for the class diagram elements. The methods of a class are placed in this symbol-table in the form $<$ *class_name* $+$

signature_name, operation implementation node >. The parameter *signature_name* is calculated by concatenating the name, the type names of the parameters and the name of the return type of the method. Following this, the method *convertCallAction()* calculates the parameter *signature_name* for the method called by the activator by concatenating the name of the operation the activator called (line 13) and the type names of the parameters this operation gets (lines $14 - 16$).

At line 17, the algorithm adds the call node representing the call action. The *if* statement that comes next checks whether this call action has a predecessor. If it does not have a predecessor, then it is the first call action belonging the activation bar and, thus, the call node should be connected to the respective operation implementation node with an edge labeled *body*. When the *if* statement at line 18 is true, the method gets the identifier of the operation implementation node from the symbol-table (the type of the classifier is concatenated with the parameter *signature_name* to get the identifier from the symbol-table) and, then, adds the edge labeled *body*. If, on other hand, the call action has a predecessor, the method gets the identifier of the call node of the preceding action as shown in lines $22 - 25$. Note that this time, the newly added action node is connected to the call node representing the preceding action by an edge labeled *next*.

DCML supports 5 kinds of call actions: instance calls, create actions, super calls, this calls and static calls. For each kind of call action, we implemented a method that handles the call action specific conversions. The method *convertCallAction()* identifies the kind of the call action and calls the appropriate converter method. The identification of the kind of a call action is done by looking at the properties of the call action or at the name and at the type (the base) of the receiving classifier. As an example, in Algorithm 9 the lines $28 - 32$ are used for identifying the instance calls and the static calls (for brevity we limited the discussion to these two kinds; nevertheless, the identification of other types is done in a similar manner). To identify the static calls, the method *convertCallAction()* compares the name of the classifier and the type of the classifier, as shown in *28*. Classifiers whose name is the same as the name of the type of the classifier represent a static class and, as a result, when the *if* statement at line 28 evaluates to true the converter calls the method that handles the conversion of the static calls. If this *if* statement evaluates to *false*, then the call is made to an instance of a class and the call is converted as an instance call. The conversions of the instance calls are implemented in the method *convertInstanceCall()*.

The call action conversion concludes by converting the arguments. The *for* statement at line 33 in Algorithm 9 traverses the list of arguments specified in the call action. A call action can pass the value of an attribute of the class or a parameter of the method to which it belongs. Execution semantics require the call action nodes to be

Algorithm 9 The pseudo-code of the method used for converting actions and call actions

```
1: method Integer convertCallAction(UMLElement callAction){
2: if actionsTable.get(callAction)! = NULL then
3:    return actionsTable.get(callAction);
4: end if
5: Integer activatorId ← actionsTable.get(callAction.getActivator());
6: if activatorId = NULL then
7:    if callAction.getActivator().getActivator().getSender() = NULL then
8:       convertInitialAction(callAction.getActivator());
9:    else
10:      convertAction(callAction.getActivator());
11:   end if
12: end if
13: String signature_name ← callAction.getActivator().getOperation().getName();
14: for parameter ∈ callAction..getActivator().getOperation().getParamList() do
15:   String signature_name ← signature_name + parameter.getType();
16: end for
17: callNodeId ← addNode("Call");
18: if callAction.getPredecessor() = NULL then
19:   operImplNodeId ← symboltable.getId(callAction.getSender().getBase() +
      signature_name);
20:   addEdge(operImplNodeId, callNodeId, "body");
21: else
22:   preActionId ← actionsTable.get(callAction.getPredecessor());
23:   if preActionId = NULL then
24:      convertAction(callAction.getPredecessor());
25:   end if
26:   addEdge(preActionId, callNodeId, "next");
27: end if
28: if callAction.getReceiver().getName() = callAction.getReceiver().getBase() then
29:   convertStaticCall(callAction, callNodeId);
30: else
31:   convertInstanceCall(callAction, callNodeId);
32: end if
33: for argument ∈ callAction.getArguments() do
34:   Integer argumentId ← symbolTable.getId(callAction.getSender().getBase() +
      argument.getName());
35:   if argumentId = NULL then
36:      argumentId ← symbolTable.getId(callAction.getSender().getBase() + signature +
      argument.getName())
37:      if argumentId = NULL then
38:         argumentId ← convertVarDecl(argument);
39:      end if
40:   end if
41:   addEdge(callNodeId, argumentId, "parameter");
42: end for
43: actionsTable.add(callAction, callNodeId);
44: return callNodeId;
45: }
```

connected to these variables for correct simulation, if they are used as arguments. To find whether the call is passing an already declared variable as an argument, the names of the arguments are searched in the symbol-table. The method first checks whether the argument is an attribute, with the call at line 34. If the argument is an attribute then its node identifier is retrieved from the symbol-table. If the argument is not an attribute, then the method checks whether it is a parameter. If the passed argument is a parameter, then its identifier is retrieved from the symbol table. The call at line 36 is used for retrieving the variable declaration node representing the parameter from the symbol table. It is important to note that the method conversion creates an entry for each parameter of a method in the symbol table in the form $<$ *class_name + signature_name + parameter_name, variable declaration node identifier* $>$. Finally, if neither of these *if* statements evaluates to *true*, then the converter treats the argument as a variable declared within the body of the method and adds a variable declaration node representing this argument, at line 38. The retrieved/created variable declaration node is connected to the call action node with the edge labeled *parameter*.

Algorithm 10 The pseudo-code of the method used for converting instance actions

```
1: method convertInstanceCall(UMLElement callAction,Integer callActionNode){
2: Integer referenceVarId ← symbolTable.getId(callAction.getSender().getBase() +
   callAction.getReciever().getName());
3: if referenceVarId = NULL then
4:    referenceVarId ← symbolTable.getId(callAction.getSender().getBase() + signature +
      callAction.getReciever().getName());
5:    if referenceVarId = NULL then
6:       referenceVarId ← addVarDecl(callAction.getReciever().getName());
7:       symbolTable.addId(callAction.getSender().getBase() + signature +
         callAction.getReciever().getName(), referenceVarId);
8:    end if
9: end if
10: addEdge(callNodeId, referenceVarId, "referenceVar");
11: Integer objectId ← valueObjectTable.get(callAction.getReciever().getName())
12: if objectId! = NULL then
13:    addEdge(referenceVarId, objectId, "instanceValue");
14:    ownerId ← valueObjectTable.get(callAction.getSender().getName().tokenize(,)[0]);
15:    addEdge(ownerId, objectId, "encapsulates");
16: end if
17: }
```

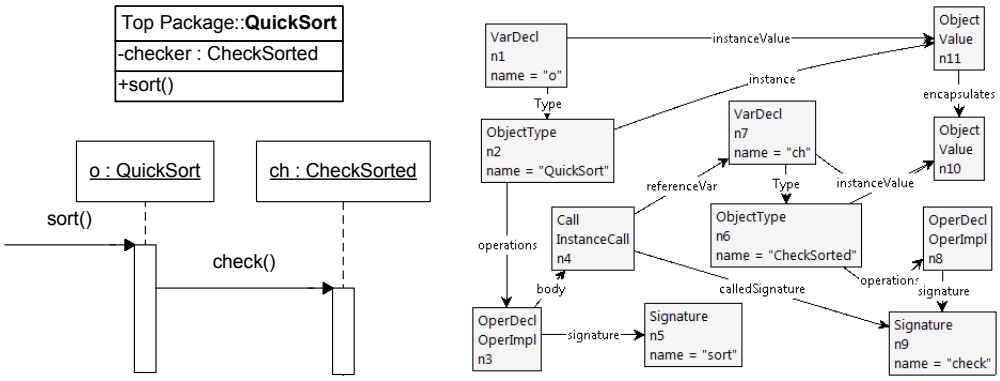
Algorithm 10 presents the method *convertinstanceCall()* to clarify how the object/value nodes resulting from the classifier conversion are used. The method first tries to locate the variable declaration node for the reference variable of the call. The method uses the name of the receiving classifier (i.e. the classifier which receives the call) as the name of the reference variable. At lines $2-5$, the symbol table is

searched to find whether the reference variable is an attribute or a parameter. If it is neither a parameter or an attribute, the method adds a new variable declaration node as shown in line 6. After converting the reference variable, the method tries the value of this variable by checking the entries of the table *valueObjectTable* at line 10. If there is an entry at this table, then the object node is retrieved and connected to the variable declaration node representing the reference variable with an edge labeled *instanceValue*, as shown in line 13. To identify the object that encapsulates the value of the reference variable, the table *valueObjectTable* is searched for an entry whose name is the same as the classifier where the call action originates from. This object node this entry holds is connected to the value of the reference variable with an edge labeled *encapsulates*. Note that the method *convertInstanceCall()* does not add a value if the first check in the table *valueObjectTable* (line 10) fails. This is because either the reference variable is not the owner of the object or the classifier is not specified.

With sequence diagrams, resolving the arguments to attributes or parameters play a crucial role in simulation. To better clarify this, we present below the DCML equivalent of various UML sequence diagrams:

- *Classifier Elements:* A classifier whose type (or base) is a primitive type is converted to a node labeled *value*. The classifiers of classes are converted to nodes labeled *object* and *value*.

- *Instance Call Action Elements with reference variables declared in the scope of the method:* Instance calls are represented by nodes labeled *Call* and *InstanceCall*. The instance call is received by the object which is hold by the reference variable of a call. The UML-to-DCML converter uses the name of the receiving classifier to identify the reference variable. In Figure A.13-(a) an example UML sequence diagram from a sorter software system with two call actions is shown: the first call is made to the method *QuickSort.sort()* and the second is made to the method *CheckSorted.check()*. The classifiers show that an instance of the class *QuickSort* called *o* received the call to the method *sort*, which, in turn, called an instance of the class *CheckSorted* called *ch*. The DCM of the sequence diagram is shown in Figure A.13-(b). Because DCML only supports communication between objects through encapsulation, the classifiers are represented as object/value nodes that are accessed through variables with the same name. For example, the object node $n11$ represents the instance of the class *QuickSort*. This instance is accessed through a variable named *o* (i.e. the owner variable of the object) which is represented by the variable declaration node $n1$. The instance of the class *CheckSort* is represented by the object node $n10$. The classifier representing the instance of the class *CheckSorted* is called *ch*, this is represented in DCML by the variable dec-

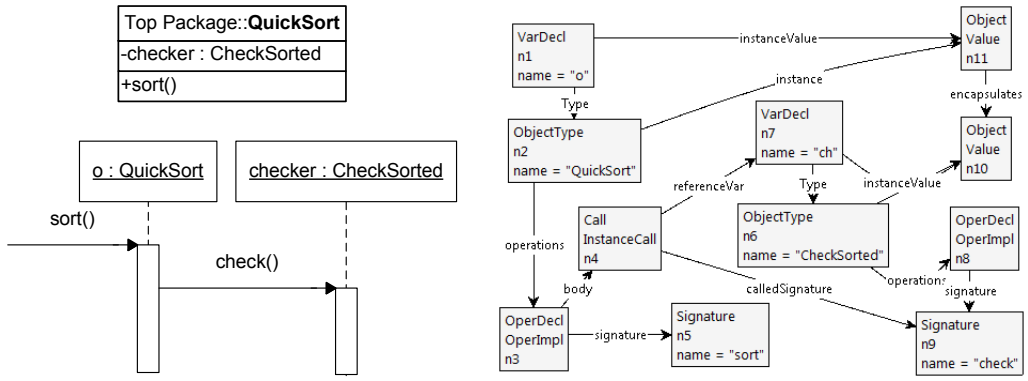


(a) (b)

Figure A.13: A sequence diagram with two instance calls (a) and the DCML representation of these instance calls (b)

laration node named *ch* whose instance value is this object (i.e. the instance of the class *CheckSorted*). The object node representing the instance of the class *QuickSort* (node $n10$) is connected to the object node representing the instance of the class *CheckSorted* with an edge labeled *encapsulates*. Combining with variables that hold these instances the classifiers of the sequence diagrams are represented as follows: the instance of the class *QuickSort* hold by the variable *o* encapsulates an instance of the class *CheckSort* and this instance is accessed through the variable *ch*. The node $n4$ represents the call action to the method *Check*. This call action is added to the body of the operation implementation node (node $n3$) representing the method *QuickSort.sort()*, because the activating call action has called the method *QuickSort.sort()* and it does not have any preceding call actions. Note that the reference variable of this call action (node $n4$) is the variable named *ch* (node $n10$). Here, because the variable *ch* is neither an attribute or a parameter of the method *QuickSort.Sort()*, the converter treated *ch* as a variable declared within the body of this method and, thus, added the variable declaration node $n4$.

- *Instance Call Action Elements with a reference variable that is an attribute of the class:* When the reference variable of a call action is an attribute of the calling class, then the edge labeled *referenceVar* should connect the call action node to the variable declaration node representing the attribute. In Figure A.14-(a), the same example sequence diagram from the sorter system is presented; however, this time the classifier showing the instance of the class *CheckSort* is named *checker*. In the class *QuickSort*, there is an attribute with the same name. This means that the call to the method *CheckSorted.check()* is referenced through the attribute *checker*. The DCML model of this sequence

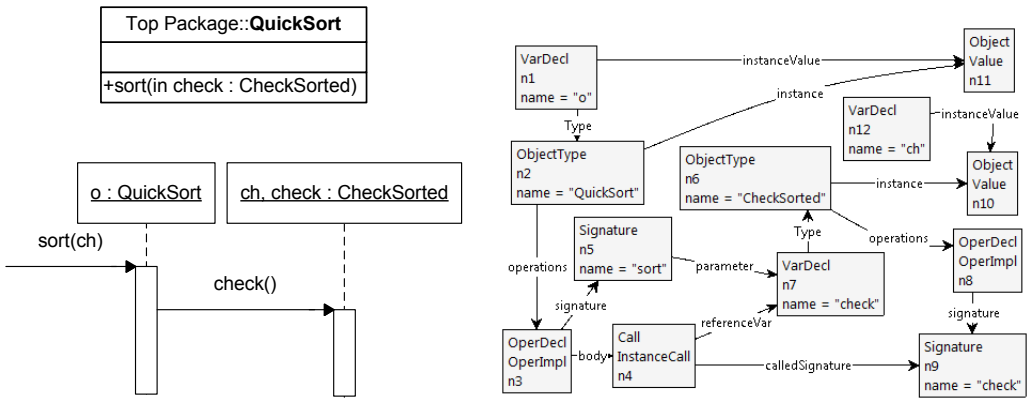


(a) (b)

Figure A.14: A sequence diagram with two instance calls where the second instance call refers to an attribute of the class *QuickSort* (a) and the DCML representation of this sequence diagram (b)

diagram is shown in Figure A.14-(b). The call action to the method *CheckSorted.check* is represented by the call node $n4$ and the attribute *checker* is represented by the variable declaration node $n7$. The call action node is connected to the variable declaration node representing the attribute *checker* with an edge labeled *referenceVar*. This shows that the call accesses the value of the attribute *checker* and, thus, the instance of the class this attribute holds will receive the call when it is simulated.

- *Instance Call Action Elements with a reference variable that is a parameter of the method:* Figure A.15-(a) shows another version of the example sequence diagram from the sorter software. In this design, the method *QuickSort.sort()* has the parameter *check* of type *CheckSorted*. The sequence diagram shows that an instance of the class *QuickSort* receives an instance of the class *CheckSorted* called *ch* when the method *sort* is called. Note that the classifier representing the instance of the class *CheckSorted* has two names: *ch* and *check*. The first name is used as an argument to the method *sort* showing that in the scope of the caller this instance is called *ch*, or, in other words, the variable *ch* is holding the instance that is passed as an argument. The second name is used in the scope of the method *QuickSort.sort()* and the instance of the class *CheckSorted* is accessed through the variable *check* in this scope. In the DCML version of the sequence diagram shown in Figure A.15-(b), the variable declaration node $n12$ represents the variable *ch* and the variable declaration node $n7$ represents the parameter *check* of the method *sort*. The call node $n4$ represents the call action to the method *CheckSorted.check()*. Because in the scope of the method *QuickSort.sort()* the name of the classifier *check* refers



(a) (b)

Figure A.15: A sequence diagram with two instance calls where the second instance call refers to a parameter of the method *QuickSort.sort()* (a) and the DCML representation of this sequence diagram (b).

to the parameter of this method, the call action node is connected with the edge labeled *referenceVar* to the variable declaration node representing the parameter *check* of the method *QuickSort.sort()*. Note that the object node $n10$ representing the instance of the class *CheckSorted* is connected as a value of the variable *ch* (node $n12$) but it is not connected to the parameter *check* (node $n7$). This is because the object is passed to the method *QuickSort.sort()* and it is owned by another object accessed through the variable *ch*. The sequence diagram does not show where the variable *ch* is declared and, thus, the converter added the variable declaration node representing this variable without connecting to a signature or to an object-type. The owner variables are an important aspect of UML-to-DCML conversions and the rule for finding out the owner variables is as follows: the variable whose name is the same as the first name of the classifier will be the owner variable and the object/value node representing the classifier will be connected to that variable.

- *Instance Call Action Elements, passing an argument that is declared with in the method:* The arguments of a call action are converted to variable declaration nodes that are connected to the node representing the call action with an edge labeled *paramValue*. In the sequence diagram of Figure A.16-(a), the call action passes the argument *toSort* to the method *CheckSorted.check()*. The classifier named *toSort* shows that this argument is an instance of the class *Vector*. The DCM of this sequence diagram is shown in Figure A.16-(b). Here, the variable declaration node $n14$ represents the argument. Following the edge labeled *instanceValue* from this variable declaration node, it can be seen that the value of this variable is the object node $n15$ representing the instance of

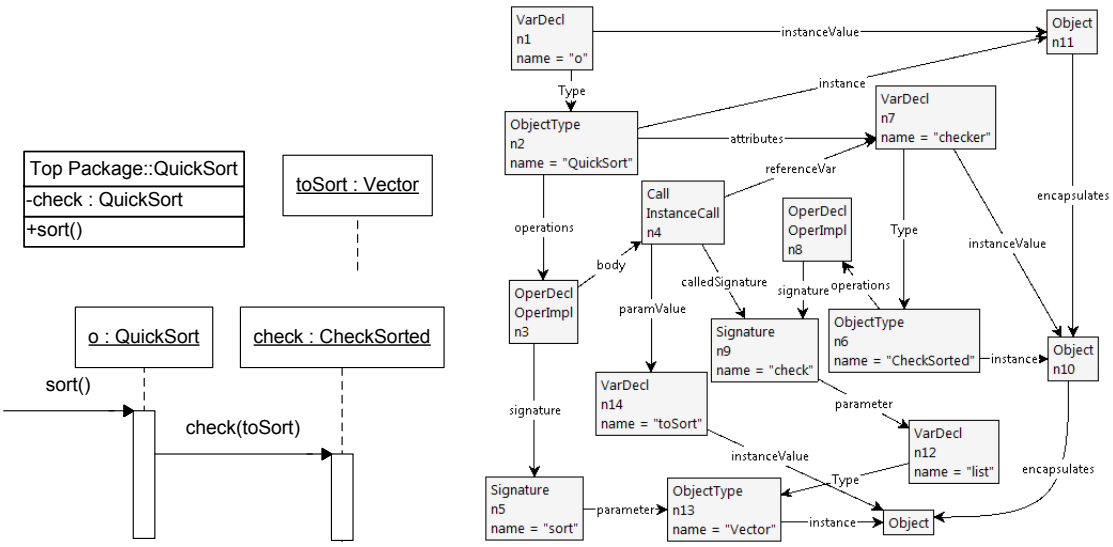


Figure A.16: A sequence diagram with a call action to the method *CheckSorted.check()* passing the argument *toSort* (a) and the DCML representation of this sequence diagram (b).

the class *Vector*. The sequence diagram in Figure A.16-(a) that the instance of the class *QuickSort* labeled *o* encapsulates the instance of the class *Vector*; the instance *toSort* first appears in the scope of the method *QuickSort.sort*. This is shown in Figure A.16-(b) with the edge labeled *encapsulates* connecting the object node $n10$ representing the instance of class *QuickSort* to the object node representing the instance of the class *Vector* in the DCM of this sequence diagram.

- *Instance Call Action Elements, passing an attribute of the class as an argument:* In Figure A.17-(a) a sequence diagram showing a call action passing the argument *toSort* is presented; however, in this design the argument *toSort* is an attribute of the class *QuickSort*. Because the call passes an attribute of the class, the call action node should be connected to the variable declaration node representing the passed attribute with the edge labeled *paramValue*. The DCML model of the sequence diagram is presented in Figure A.17-(b). Here, the variable declaration node $n14$ represents the attribute *toSort* as this node is connected to the object-type node $n2$ representing the class *QuickSort* with the edge labeled *attributes*. The call node $n4$ represents the call action to the method *CheckSorted.check()*. To show that this call action passes the attribute as an argument, it is connected to the variable declaration node $n14$ with an edge labeled *paramValue*.

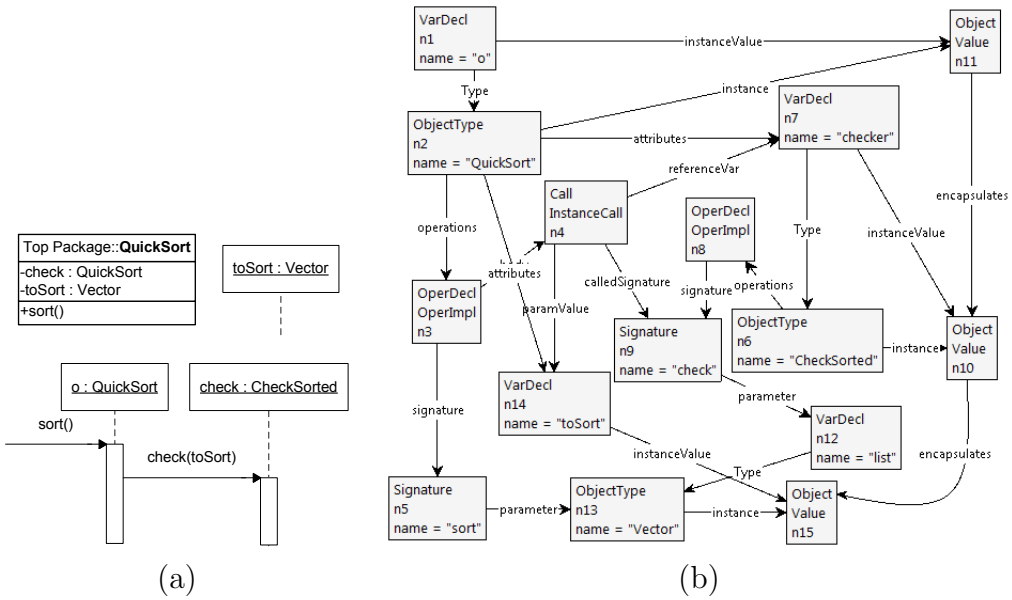


Figure A.17: A sequence diagram with the call action to the method *CheckSorted.check()* passing the argument *toSort* (a) and the DCML representation of this sequence diagram (b).

- *Instance Call Action Elements, passing a parameter of the method as an argument:* In the sequence diagram shown in Figure A.18-(a), the first call action passes the argument *list* to the method *QuickSort.sort()*. Upon receiving this call, the instance of the class *QuickSort* calls the method *CheckSorted.check()*. This call passes the argument *toSort* which is specified as a parameter of the method *QuickSort.sort()*; so, the value of the parameter *toSort* is passed to the method *CheckSorted.check()*. It can be seen from the first call action, that the value of the parameter *toSort* is the instance of the class *Vector* shown with the classifier *list*. The DCML model of the sequence diagram is presented in Figure A.18-(b). The call node $n4$ represents the second call action, the call action to the method *CheckSorted.sort()*. The variable declaration node $n14$ represents the parameter *toSort* and, because the call action passes this parameter as an argument, the call action node $n4$ is connected to the this variable declaration node with the edge labeled *paramValue*. The object node $n15$ represents the instance of the class *Vector*. Note that this object node is connected to the parameter *toSort* because the variable *toSort* is not the owner variable. Instead, the object node is connected to the variable declaration node $n16$ named *list*, because this variable is the owner variable.

- *Return Action Elements with no return value:* The return actions are in DCML represented by nodes labeled *return*. In Figure A.19-(a) the return actions are

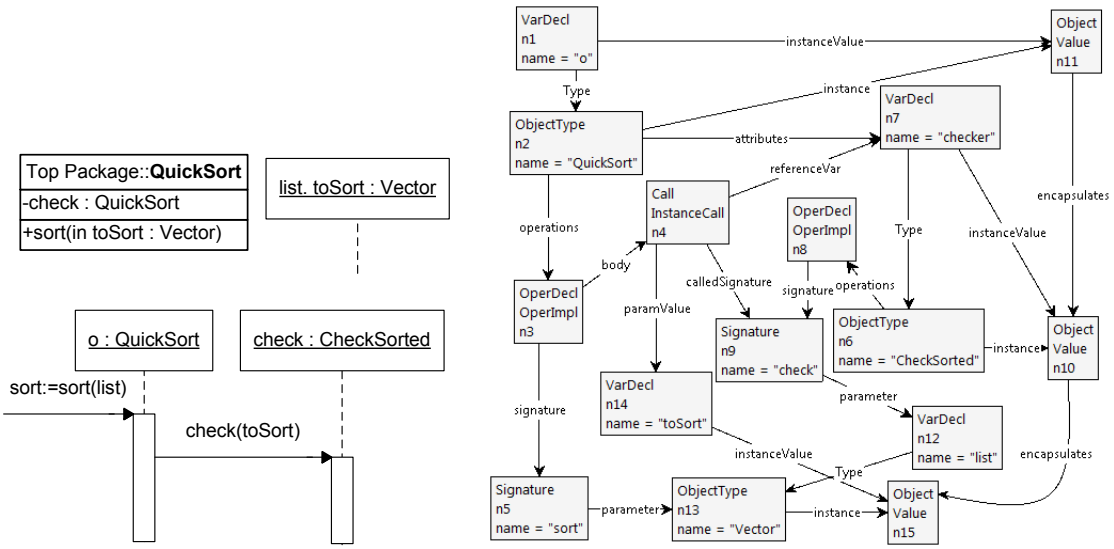


Figure A.18: A sequence diagram with the call action to the method *CheckSorted.check()* passing the parameter *toSort* of the method *QuickSort.sort()* as an argument (a) and the DCML representation of this sequence diagram (b).

shown for the sorter system example. These return actions do not return a value as there is no text describing the return value on the actions. Figure A.19-(b) shows this sequence diagram in DCML, where the return actions are represented by the nodes $n17$ and $n18$. The return action node $n18$ is connected to the operation implementation node $n8$ representing the method *CheckSorted.check()* with edge labeled *body* because this action does not have a predecessor (i.e. it is first action in the activation bar). The call action to the method *CheckSorted.check()* precedes the return action of the method *QuickSort.sort()*; thus, the return action node $n17$ is connected to the call node representing this call action with an edge labeled *next*.

- *Return Action Elements with return value:* A return value is modeled by connecting the return action node to the variable declaration node holding the value to be returned with an edge labeled *returnVal*. Figure A.20-(a) shows the sequence diagram where the call to the method *CheckSorted.check()* returns the value shown with the classifier *checkResult*. In the DCML model of this sequence diagram, shown in Figure A.20-(b), the value node $n21$ and the variable declaration node $n20$ represent this classifier. The return node $n18$ representing the return action from the method *CheckSorted.check()* is connected to the variable named *checkSort* to show that the value of this variable in the executing object will be returned. Since the owner variable of the

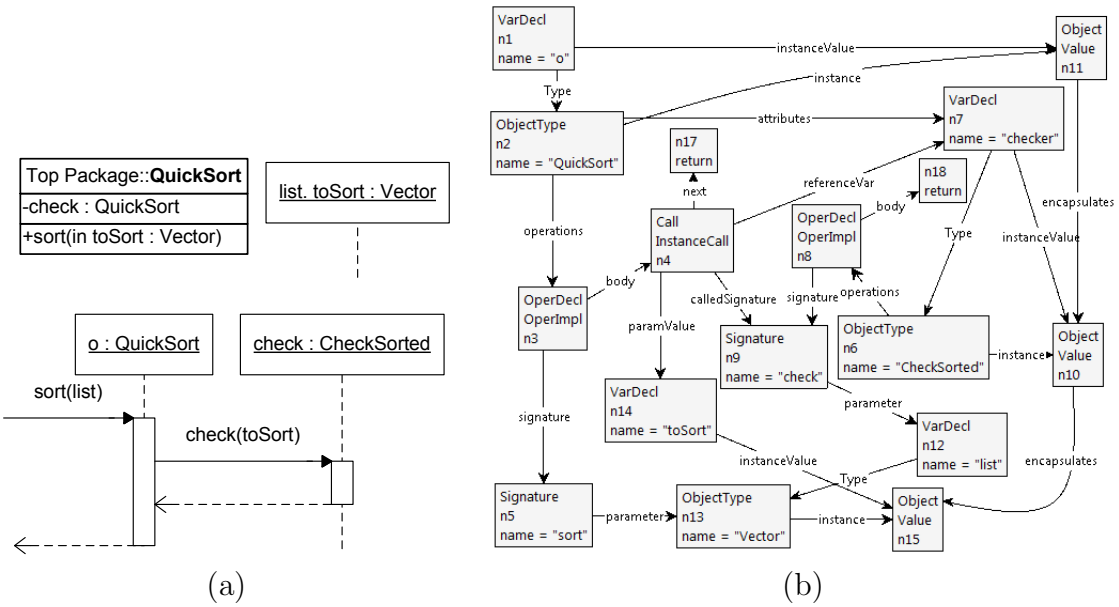
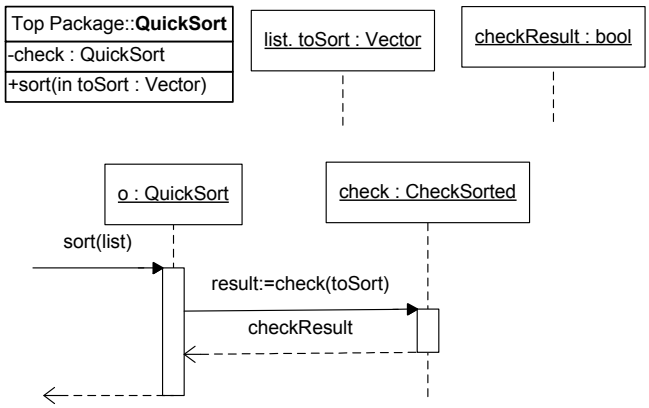
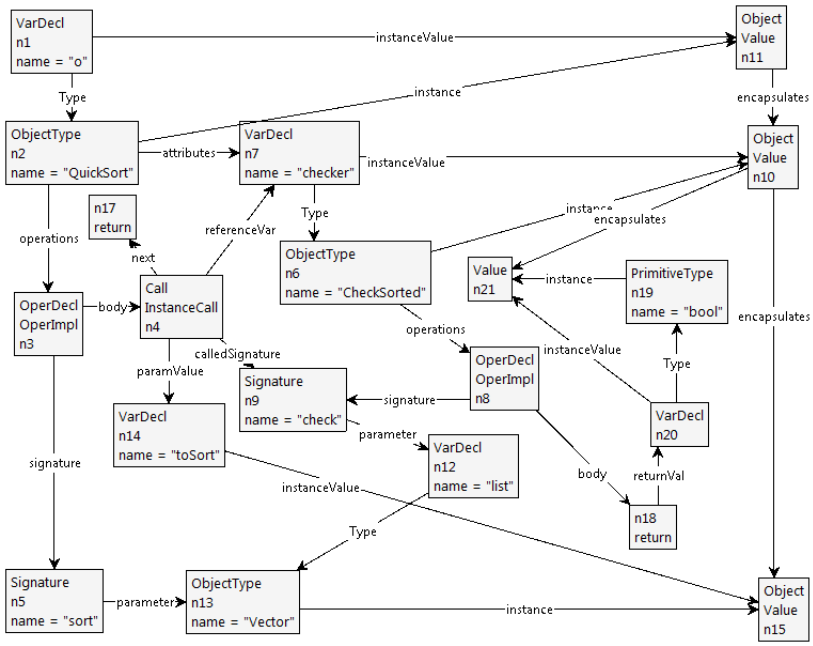


Figure A.19: A sequence diagram of Figure A.18-(a) with the return actions (a) and the DCML representation of showing the return actions (b).

classifier *checkSort* is the variable with the same name, the value node $n21$ is connected to the variable node $n20$. The sequence diagram shows that at the instance *check* of class *CheckSorted* the variable *checkSort* is a boolean value. Thus, in the DCML model of this sequence diagram, the object node $n10$ representing the instance of the class *CheckSorted* is connected with the edge labeled *encapsulates* to the value node representing the classifier *checkSort*.



(a)



(b)

Figure A.20: A sequence diagram of Figure A.18-(a) with the return actions (a) and the DCML representation of showing the return actions (b).

Appendix B

The Raw Data of the Experiments

This section presents the raw data for the experiments described in Chapter 4

B.1 Raw Data of the Experiment for CTL Based Feedback Mechanism

The experiment on the effectiveness of the CTL based feedback mechanism was conducted over 46 students. Due to errors, the data from two students from the tool support group is discarded. Table B.1 presents the raw data of this experiment; the students whose data are discarded are shown with N/A.

B.2 Raw Data of the Experiment for Control Automaton Based Feedback Mechanism

Two experiments are conducted at the same time to test the effectiveness of the feedback mechanism based on control automaton. The first experiment (E_1) used VSL specifications that tested existence of a reconfiguration and the second experiment (E_2) used VSl specifications that tested violations of reconfiguration invariants. The experiments are conducted at the same time; students are divided into two groups: 1) manual evaluation for E_1 and tool supported evaluation for E_2 2) tool supported evaluation for E_1 and manual evaluation for E_2. In the remaining sections, the raw data of the both experiments are provided.

Table B.1: The raw data for the experiment to test the effectiveness of the CTL based feedback mechanism

Tool Support		Manual Evaluation	
Student	# of Errors	Student	# of Errors
1	1	24	2
2	2	25	3
3	1	26	1
4	N/A	27	2
5	1	28	2
6	2	29	1
7	3	30	3
8	1	31	3
9	2	32	2
10	1	33	1
11	1	34	1
12	1	35	3
13	1	36	1
14	1	37	1
15	1	38	3
16	N/A	39	1
17	1	40	1
18	1	41	3
19	0	42	2
20	1	43	3
21	3	44	3
22	3	45	3
23	1	46	0

Table B.2: The raw data for experiment E_1

	E_1		E_2	
Student	# of Errors	Group	# of Errors	Group
1	0	Tool	1	Manual
2	0	Tool	1	Manual
3	0	Tool	2	Manual
4	N/A	Tool	N/A	Manual
5	0	Tool	1	Manual
6	0	Tool	0	Manual
7	0	Tool	2	Manual
8	0	Tool	0	Manual
9	0	Tool	2	Manual
10	2	Manual	0	Tool
11	1	Manual	0	Tool
12	N/A	Manual	N/A	Tool
13	N/A	Manual	N/A	Tool
14	0	Manual	0	Tool
15	1	Manual	0	Tool
16	1	Manual	0	Tool
17	2	Manual	0	Tool
18	2	Manual	0	Tool
19	1	Manual	0	Tool

B.2.1 Experiment E_1

19 students have entered the experiment; the data for 3 students are discarded due to errors (students *4,12,13*). Table B.2 columns *2-3* presents the raw data of the experiment used for the statistical analysis.

B.2.2 Experiment E_2

Table B.2 columns *4* and *5* presents the raw data for E_2.

Appendix C

The Language for specifying UML class and sequence diagrams

In this section, the textual syntax of representing UML diagrams are described. In the experiments (See Section 4.3), the UML diagrams of the industrial software system is expressed using this syntax. The students have received this textual version of the diagrams and they also made modifications on the textual version.

We used the Textual Concrete Syntax (TCS) [75] to generate textual UML-to-DCML converter tools. TCS is a text-to-model (model-to-text) converter program generator for meta-models defined using the KM3 language [74]. In our implementation, we expressed the DCML meta-model (a subset of the UML meta-model) in the KM3 language and defined a textual syntax for this subset with TCS. The converter program generated by TCS, converts the textual version of the UML diagrams to ECORE models; we programmed another converter that converted these ECORE models to graphs in DCML.

Because TCS does not support scopes, the called signatures and the referenced variables are not resolved in the resulting model. For example, a method call like *a.foo* in DCML is modeled by connecting the node representing the method call to the node representing the declaration of the variable *a*. This requires a trace in the scope to find the where variable *a* is declared. TCS allows model elements to added to context; thus, an element in the context can be referred by its attributes. The limitation here is that TCS only supports a global context where, for example, if two variables with the same name are declared under different classe,s TCS cannot resolve the referenced variable.

To address this problem, the names of the referenced attributes and signatures are stored as attributes. The ECORE models to graph converter keeps track of the

scopes and converters these attributes to references. In the rest of this chapter, we detail the syntax of the textual language for UML (TUML).

C.1 Diagram Container

The diagram container is a model element that contains the class and the sequence diagram classifier definitions from which the DCML model is generated. The diagram container is not the DCML meta-model; though, it is specified in the KM3 specification of DCML to put together the UML classes and classifiers needed to generate the model in DCML. This element is ignored when the model is loaded to GROOVE.

The diagram container is the root element and it is specified in KM3 as follows:

```
1: class UMLDiagramContainer extends LocatedElement{
2: reference types [1-*] container : Type;
3: reference instance [1-*] container : Classifier;
4: reference values[*] container: Value;
5: reference startingFrame container : OperFrame;
6: }
```

As can be seen from this specification, the diagram container consists of at least one type specification (classes, interfaces and/or primitive types) called *Type* (line 2) and at least one classifier specification called *Classifier* (line 3). The diagram container has a third type of element called *Value*. With these elements, values for attributes or method returns can be specified.

DCML contains operation frames, which are used for resolving the values of the variables. The operation frames are not included in sequence diagrams and an initial operation frame is needed to mark the action from which the simulation starts. The UML-to-DCML converter implement in AgroUML marks the first action of the user-specified sequence diagram and adds the operation frame accordingly. With the textual version of the UML diagrams, on the other hand, we added the ability to specify the first action. This specification is the fourth element in the diagram container; the type of this element is an operation frame and it has the name *startingFrame*.

The syntax for the diagram container is specified by the keywords *UML Diagram-Container* { followed by the type specifications, the classifier specification, the value specification and, lastly, the starting operation frame specification.

C.2 Object Type Specification

In DCML classes and interfaces are called object types. An object type consists of attributes and operations; below is the syntax for object types:

```
1: template ObjectType
2: : “Class” (final ? “final”) (abstract ? “abstract”) (interface? “interface”) name
   (isDefined(superType)? “extends” superType {refersTo = name})“{”
3: attributes
4: operations
5: “}”;
```

Here, the second line describes how an object type is declared in TUML. The declaration of an object type starts with the keyword *Class*. If the type is an interface then it is followed with the keyword *interface*. If the type is a class, then the keyword *Class* is followed by two optional keywords that describe the properties of the class. For abstract classes this keyword is followed by the keyword *abstract* and for final classes the keyword *Class* is followed by the keyword *final*. After these, the name of the object type is specified. The super-types of an object type is specified by the keyword *extends* and, then, the names of the super types are listed.

The declaration of object type is followed by its attribute declarations and operation declarations.

C.2.1 Attribute Declarations

The attributes of classes are variable declarations which are specified as the name of the variable followed by the name of the type of the variable. The syntax of variable declarations are defined in TCS as follows:

```
template VarDecl
: name ”:” type {refersTo = name}
;
```

C.2.2 Method Declarations

In UML there two kinds of operations: abstract methods and implemented methods. The syntax of an abstract method is specified follows:

```
template AbsMethod
: “abstract” “operation” signature
;
```

Here, the abstract method declaration starts with the keywords *abstract* and *operation* and it is followed by the signature of the operation.

The syntax of implemented method is given below:

```
template ImplMethod
: "operation" "<" id ">" (final ? "final") (static ? "static") signature
;
```

The specification of an implemented method starts with the keyword *operation* followed by a user specified identifier (*id*). In UML diagrams, the body (or the life-line) of the operation is specified in classifiers of the sequence diagrams. We follow this notation in TUML and use the user specified identifiers to match the body in the classifier specification with the operation implementation specified in the object type declaration. The identifier is followed by two optional keywords used for describing the properties of the operation: for static operations the keyword *static* is used and for final operations the keyword *final* is used. The specification of an operation implementation is finished with the signature of the operation.

A special type of operation implementation is the constructor. A constructor specification starts with the keyword *constructor*. Similar to the operation implementation declaration, this keyword is followed by a user specified id and, then, the signature of the constructor.

The syntax of a signature is defined as follows:

```
template Signature
: name "("(isDefined(parameter) ? parameter {separator = ","}) ")" (isDefined(returnType) ? ":" returnType {refersTo = name})
;
```

Here, *name* is a string, which refers to the name of the signature. The name is followed by parenthesis. If the signature has parameters, then the parameters are specified within the parenthesis. Parameters are separated by commas. The syntax of the parameter is the same as the syntax used for variable declaration. It is important to note that every parameter definition creates a new variable declaration node.

The return variable (and value) is optionally specified. If an operation has a return variable, then the closing parenthesis is followed by a colon and the name of the type.

C.3 Value Specification

The value of a variable can be specified in UML as a property of the attribute or parameter of the function. Because of this, in TUML we decided to place them separately from the classifier and class definitions.

Bellow the KM3 specification of the element *value* is provided:

```
1: class Value extends LocatedElement {
2: attribute created : Boolean;
3: attribute id : String;
4: attribute nodeId : String;
5: reference type : Type oppositeOf instance;
6: attribute val : String;
7: attribute accessed : String
8: reference owner [0-1] : Object oppositeOf encapsulates;}
```

The KM3 version of the element *value* differs in three ways than the value nodes of DCM. The first difference is the way how the type of the value is specified. In DCM, the type of value is identified with an edge, labeled *instance*, from the type node to the value node. This makes it hard to textually specify the type of a value; so, instead we use an edge called *type* from the value node to the type node. This edge, in line 5, is defined as the opposite of the edge labeled *instance*.

The second difference is in the *encapsulates* edge. The encapsulates edge is drawn from an object node to a value node in DCML. This requires a syntax where the object is specified with all objects it encapsulates; this may be hard, because the encapsulated objects can change as the designer progresses on the sequence diagram. We addressed this with the edge *owner* which is defined as the opposite of the edge *encapsulates* (line 8). For example, a value is declared to be owned by an object with the *owner* edge when the value is specified.

The third difference is that the edge *instanceValue* connecting a variable declaration to a value is not included, due to the limited scoping support in TCS. Instead, the attribute *accessed* holds the name of the variable. When the model is loaded, the variable this name refers to is resolved and the edge *instanceValue* is added.

We defined the syntax of value nodes as follows:

```
template Value
: "<" id ">" accessed ":" type refersTo = name "=" val
(isDefined(owner) ? "owner" "<" owner refersTo = id ">" ) (created? "created")
;
```

The value specification starts with a user given identifier; this identifier is used to distinguish the value from other values. The identifier is followed by the name of the variable which holds this value and the type of the value. The type is followed by the name of the value. In DCML, it is possible to specify a name for a value; so that, how the value is passed between different objects can be traced. If the value has an owner (i.e. an object encapsulates this value) then this is specified with the keyword *owner* followed by the identifier of the encapsulating object (objects are specializations of values in DCML so they also have identifiers). If the value is not owned, then it is created; this it is specified by the keyword *created*.

C.4 Classifier Specification

A classifier of a sequence diagram shows which object has received calls and the life-lines that show what the object does when it has received a call. Following this, a classifier consists of an object and a series of life-lines in the textual version of the UML. Below, the syntax for the classifier is shown.

```
template Classifier addToContext
: "<" id ">" (isDefined (accessed) ? accessed) ":" type
refersTo = name (isDefined(owner) ? "owner" "<" owner refersTo = id ">")
(created? "created") "{"
(isDefined(lifeline) ? lifeline)
"}";
```

The second line in the syntax above is the object specification. The object specification is similar to the specification of a value (Section C.3). This specification is followed by the specification of the life-lines; a classifier can have zero or more life-lines because a life-line is drawn for each method call the object receives.

The life-line specification starts with an identifier. This identifier should be same as the id of the operation implementation from which the object has received the call. The identifier is followed by braces and within these braces the actions of the life-line are placed. In the remainder of the section, the syntaxes of these actions are detailed.

C.4.1 Call Action

The specification of a call action starts with the keyword *call*. This keyword is followed by the name of the reference variable of the call and the name of the signature. The actual model elements these names refer to are not resolved by TCS,

these names are stored as attributes of the call element. These are followed by parentheses. If the call passes parameters, these parameters are listed within the parentheses. The syntax of the parameters are the same as the syntax of the variable declarations and the parameters are separated with commas. Below the syntax for the call action is presented:

```
template CallAction
: (isDefined(label) ? label) "call" referenceVar "."called"("
(isDefined(passes) ? passes {separator = ","}) ")"
(isDefined(assignedVar) ? "="assignedVar)
(isDefined(next) ? next);
```

In sequence diagrams, it is possible to specify the variable into which the value returned by the call is stored. For example, a call action can be specified *a=f.foo()*, where *a* is a variable that stores the value returned by the method *foo()*. In the syntax presented above, the forth line shows the syntax of how this is specified in TUML. Here, if the method's return value gets assigned to a variable, then the parenthesis is followed by an equal sign and the name of the variable that gets the return value.

The syntax for self-calls and super-calls is similar to the syntax of a call action. The difference is that for these calls the reference variable is not specified. Instead of the reference variable, the keyword *this* is used for self-calls and the keyword *super* is used for super-calls.

C.4.2 Create Action

The create action specification starts with the keyword *new* and is followed by the name of the signature of the constructor. The syntax of the parameter list and the variable that receives the created object is the same as syntax of those elements in call actions.

C.4.3 Return Action

The syntax of the return action starts with the keyword *return* and, if the method has a return value, followed by the name of the variable whose value is returned.

C.4.4 Conditional Frame

The syntax of the conditional frame is:

template ConditionalFrame
: (isDefined(label) ? label) “ConditionalFrame” “(” condition_on “)” “{”
possible_next “}” (isDefined(next) ? next);

The conditional execution specification starts with the keyword *ConditionalFrame*, followed by the name of the variable on which the condition is defined. This is followed by the frame fragments (labeled *possible_next*). If the conditional frame is an optional frame (*opt*), then only fragment is placed in it (the action that comes after the conditional frame) is specified. If the conditional frame is an alternative frame, then, more than one fragment is placed in it.

The specification of a fragment starts with a string defining the condition under which the fragment is executed and followed by the actions within the fragment.

C.4.5 Dynamic Type Loading

Dynamic type loading is specified with a similar syntax as for optional frames. The specification starts with the keyword *DynamicTypeLoader*, followed by braces. Within these braces a frame fragment is placed. This frame fragment shows the actions that are executed when the type is loaded successfully.

Optionally, the specification can be followed by the keyword *DynamicTypeLoader_fail* and braces. Within these braces, the actions that are executed when the loading of the dynamic type fails are specified.

C.4.6 Polymorphic Reconfiguration

Polymorphic reconfiguration is specified with the keyword *polymorphicReconfiguration* followed by the call action.

C.5 Operation Frame

The UML diagrams to DCML converter selected the first action of the first sequence diagram as the entry point of the simulation and created the operation

frame according to this action. We added a separate section to add separate section (after the classifiers) in which the user can specify the operation frame from with the simulation starts from. Below, the syntax for the operation frame is given:

template OperFrame
: self refersTo = id ":" executingType {refersTo = name}
"executes" executes {refersTo = label};

Here, the specification starts with the id of the object that is executing and the id is followed by the name of the type whose instance is executing. The specification of an action (like a call action) starts with an optional label. This label is used in the operation frame specification to specify the action from which the simulation starts. The name of the type is followed by the keyword *executes* and the label of action to specify this action.

Bibliography

[1] Antlr parser generator. http://www.antlr.org/.

[2] Argo uml [online] http://argouml.tigris.org.

[3] Borland together [online] http://www.borland.com/us/products/together/index.html.

[4] Cvs: [online] http://www.cvs.org.

[5] Darwin project [online], http://www.embeddedsystems.nl/site/projects/darwin.html.

[6] Eclipse: [online] http://www.eclipse.org.

[7] Gace: Graph-based adaptation, configuration and evolution modeling [online] http://trese.cs.utwente.nl/willevolve/.

[8] Gnu prolog. http://www.gprolog.org/.

[9] Java language specification [online] http://java.sun.com/docs/books/jls/third_edition/html/j3toc.html.

[10] Rational rose [online] http://www-01.ibm.com/software/awdtools/developer/rose/.

[11] Spss data analysis software. www.spss.com.

[12] Visual studio: [online] http://msdn.microsoft.com/en-gb/vstudio/default.aspx.

[13] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *ASE*, pages 271–275, CA, USA, 2002. IEEE.

[14] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[15] L. Apvrille, P. De Saqui-Sannes, P. Sénac, and C. Lohr. Verifying service continuity in a dynamic reconfiguration procedure: Application to a satellite system. *Automated Software Engg.*, 11(2):167–191, 2004.

[16] Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. Turtle: A real-time uml profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7):473–487, 2004.

[17] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[18] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *ICSE '04*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE '06*, pages 72–81, 2006.

[20] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Sys. J.*, 15(1):225–252, 1976.

[21] PerOlof Bengtsson and et al. Architecture-level modifiability analysis (alma). *Jour. of Sys. and Soft.*, 69(1-2):129–147, 2004.

[22] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 511–520, New York, NY, USA, 2008. ACM.

[23] Gilad Bracha. Pluggable type systems. *OOPSLA workshop on revival of dynamic languages*, 2004.

[24] Jeremy S. Bradbury and et al. A survey of self-management in dynamic software architecture specifications. In *WOSS '04*, pages 28–33, 2004.

[25] Antonio Bucchiarone and Juan P. Galeotti. Dynamic software architectures verification using dynalloy. *ECEASST*, 10, 2008.

[26] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.

[27] C#. C# language specification v3 [online].

[28] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software evolution through dynamic adaptation of its oo design. In *Objects, Agents and Features*, pages 69–84. Springer-Verlag, jul 2004.

[29] Mel Ó Cinnéide and Paddy Nixon. Automated software evolution towards design patterns. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165, NY, USA, 2001. ACM.

[30] S. Ciraci, W. K. Havinga, M. Akşit, C. M. Bockisch, and P. M. van den Broek. A graph-based aspect interference detection approach for uml-based aspect-oriented models. Technical Report TR-CTIT-09-39, Enschede, September 2009.

[31] Selim Ciraci, Pim van den Broek, and Mehmet Aksit. A taxonomy for a constructive approach to software evolution. *JSW*, 2(2):84–97, 2007.

[32] Selim Ciraci, Pim van den Broek, and Mehmet Aksit. Framework for computer-aided evolution of object-oriented designs. *COMPSAC*, pages 757–764, 2008.

[33] Selim Ciraci, Pim van den Broek, and Mehmet Aksit. Graph-based verification of static program constraints. *to appear in SAC*, 2010.

[34] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[35] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[36] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 89–108, New York, NY, USA, 2006. ACM.

[37] D. R. Cok and J. R. Kiniry. Esc/java2: Uniting esc/java and jml. In *CASSIS'04*, pages 108–128. Springer, 2004.

[38] Michael A. Covington, Donald Nute, and André Vellino. *Prolog programming in depth*. Scott, Foresman & Co., Glenview, IL, USA, 1987.

[39] Roger F. Crew. Astlog: a language for examining abstract syntax trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[40] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08*, pages 171–180, New York, NY, USA, 2008. ACM.

[41] A. Del Bimbo, L. Rella, and E. Vicario. Visual specification of branching time temporal logic. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 61–68, Sep 1995.

[42] Jens Dietrich and Chris Elgar. A formal description of design patterns using owl. *aswec*, 0:243–250, 2005.

[43] D. S. Distefano, J. P. Katoen, and A. Rensink. Who is pointing when to whom? In *FSTTCS*, volume 3328 of *Lecture notes in Computer Science*, pages 250–262. Springer, 2004.

[44] L. Dobrica and E. Niemel. A survey on software architecture analysis. *IEEE Transactions on Software Engineering*, 28:638–653, July 2002.

[45] Amnon H. Eden. A theory of object-oriented design. *Information Systems Frontiers*, 4(4):379–391, 2002.

[46] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69, 1979.

[47] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08*, pages 391–400, New York, NY, USA, 2008. ACM.

[48] G. Engels, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to operational semantics of behavioral diagrams in uml. In *OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, 1999.

[49] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of uml activities using dynamic meta modeling. In *FMOODS'07*, LNCS, pages 76–90. Springer-Verlag, 2007.

[50] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[51] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, 1994.

[52] Paolo Falcarin and Gustavo Alonso. Software architecture evolution through dynamic aop. In *EWSA*, pages 57–73, 2004.

[53] F. Faul, E. Erdfelder, A.-G. Lang, and A. Buchner. G*power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods*, 39:175–191, 2007.

[54] Bernd Fischer and Eelco Visser. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 239–253. Spinger-Verlag, 2004.

[55] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition*. Addison-Wesley Professional, 2003.

[56] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[57] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. Dynalloy: upgrading alloy with actions. In *ICSE '05*, pages 442–451, New York, NY, USA, 2005. ACM.

[58] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: from concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[60] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. *SIGSOFT Softw. Eng. Notes*, 29(6):179–188, 2004.

[61] Todd L. Graves, Alan F. Karr, J.s. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[62] Roy Gronmo, Birger Moller-Pedersen, and Goran K. Olsen. Comparison of three model transformation languages. In *ECMDA-FA '09*, pages 2–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[64] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3-4):287–313, 1996.

[65] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. Codequest: querying source code with datalog. In *OOPSLA '05*, pages 102–103, New York, NY, USA, 2005. ACM.

[66] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. pages 10 pp.–150, 0-0 2006.

[67] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. *SIGPLAN Not.*, 36(5):13–23, 2001.

[68] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98*, pages 69–72, NY, USA, 1998. ACM.

[69] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling collaborations with dynamic structural adaptation in mechatronic uml. In *SEAMS '08*, pages 33–40, New York, NY, USA, 2008. ACM.

[70] D. Hou and H.J. Hoover. Using scl to specify and check design intent in source code. *Software Engineering, IEEE Transactions on*, 32(6):404–423, June 2006.

[71] Javier Luis Cánovas Izquierdo and Jesús García Molina. A domain specific language for extracting models in software modernization. In *ECMDA-FA'09*, pages 82–97, 2009.

[72] Praveen K. Jayaraman and Jon Whittle. Ucsim: A tool for simulating use case scenarios. In *ICSE COMPANION '07*, pages 43–44, Washington, DC, USA, 2007. IEEE Computer Society.

[73] W. L. Johnson and M. Feather. Building an evolution transformation library. In *ICSE '90*, pages 238–248, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[74] Frédéric Jouault and Jean Bézivin. Km3: A dsl for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*, LNCS, pages 171–185, Berlin, 2006. Springer.

[75] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06*, pages 249–254, New York, NY, USA, 2006. ACM.

[76] H. Kastenberg, A. G. Kleppe, and A. Rensink. Defining oo execution semantics using graph transformations. In *8th IFIP*, volume 4037 of *LNCS*, pages 186–201, 2006.

[77] H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In *SPIN'06*, volume 3925, pages 299–305, Berlin, 2006. Springer-Verlag.

[78] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM '01*, page 736, Washington, DC, USA, 2001. IEEE Computer Society.

[79] R. Kazman and et al. The architecture tradeoff analysis method. *iceccs*, 00:0068, 1998.

[80] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. Saam: a method for analyzing the properties of software architectures. In *ICSE '94*, pages 81–90, CA, USA, 1994. IEEE.

[81] J. Keeney and V. Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. pages 3–14, 2003.

[82] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Softw. Eng.*, 25(4):493–509, 1999.

[83] Dong Kwan Kim and Eli Tilevich. Overcoming jvm hotswap constraints via binary rewriting. In *HotSWUp '08*, pages 1–5, New York, NY, USA, 2008. ACM.

[84] T. Kobayashi and M. Saeki. Software development based on software pattern evolution. *APSEC '99*, pages 18–25, 1999.

[85] Jun Kong, Kang Zhang, Jing Dong, and Dianxiang Xu. Specifying behavioral semantics of uml diagrams through graph transformations. *J. Syst. Softw.*, 82(2):292–306, 2009.

[86] Sabine Kuske. A formal semantics of uml state machines based on structured graph transformation. In *UML'01*, pages 241–256, London, UK, 2001. Springer-Verlag.

[87] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[88] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *ARM '07*, pages 1–6, New York, NY, USA, 2007. ACM.

[89] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[90] M. M. Lehman, D. E. Perry, J. C. F. Ramil, W. M. Turski, and P. Wernick. Metrics and laws of software evolution. In *Fourth International Symposium on Software Metrics*, pages 20–32, November 1997.

[91] Xia Liu and Qing Wang. Study on application of a quantitative evaluation approach for software architecture adaptability. *QSIC 2005*, pages 265–272, 2005.

[92] Christopher M. Lott and Dieter H. Rombach. Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering*, 1(3):241–277, January 1996.

[93] Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan, and Rick Kazman. An approach to software architecture analysis for evolution and reusability. In *CASCON '97*, page 15. IBM Press, 1997.

[94] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00*, pages 337–361, London, UK, 2000. Springer-Verlag.

[95] Edward McCormick and Kris De Volder. Jquery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA, 2004. ACM.

[96] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications*, 23(4):405 – 413, 2002.

[97] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 2008.

[98] Tom Mens and Amnon H. Eden. On the evolution complexity of design patterns. *Electronic Notes in Theoretical Computer Science*, 127(3):147–163, April 2005.

[99] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[100] Tom Mens, Ragnhild Van Der Straeten, and Maja DHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model Driven Eng. Lang. and Sys.*, volume 4199/2006, pages 200–214, 2006.

[101] Tom Mens and Tom Tourw. A declarative evolution framework for object-oriented design patterns. *Software Maintenance, IEEE International Conference on*, 0:570, 2001.

[102] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[103] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MoDELS '08*, pages 782–796, Berlin, Heidelberg, 2008. Springer-Verlag.

[104] Jörg Niere and et al. Towards pattern-based design recovery. In *ICSE '02*, pages 338–348, 2002.

[105] M. O'Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 463, Washington, DC, USA, 1999. IEEE Computer Society.

[106] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[107] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[108] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[109] A. Rensink. Representing first-order logic using graphs. In *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Berlin, 2004. Springer Verlag.

[110] A. Rensink and T. Staijen. Controlled rule application using failure automata.

[111] Arend Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Trans. with Industrial Relevance*, pages 479–485. Springer-Verlag, 2004.

[112] Arend Rensink and Anneke Kleppe. On a graph-based semantics for uml class and object diagrams. *ECEASST'08*, 10, 2008.

[113] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.

[114] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01*, pages 46–53, New York, NY, USA, 2001. ACM.

[115] M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 97–106, Sept.-1 Oct. 2005.

[116] M. Shaw. Heterogeneous design idioms for software architecture. In *Software Specification and Design, 1991., Proceedings of the Sixth International Workshop on*, pages 158–165, Oct 1991.

[117] Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC-FSE '07*, pages 175–184, New York, NY, USA, 2007. ACM.

[118] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with kaba. In *OOPSLA '04*, pages 315–330, New York, NY, USA, 2004. ACM.

[119] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[120] Arie van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. 1996.

[121] Eelco Visser. Program transformation with stratego/xt. *Domain-Specific Program Generation*, 30(16):216–238, 2004.

[122] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831 – 873, 2005.

[123] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[124] M. Vittek. Refactoring browser with preprocessor. In *CSMR'03*, pages 101–110, March 2003.

[125] Michel Wermelinger and José Luiz Fiadeiro. Algebraic software architecture reconfiguration. *SIGSOFT Softw. Eng. Notes*, 24(6):393–409, 1999.

[126] Jon Whittle. Transformations and software modeling languages: Automating transformations in uml. In *UML '02*, pages 227–242, London, UK, 2002. Springer-Verlag.

[127] Jon Whittle. Precise specification of use case scenarios. In *FASE'07*, volume 44 of *LNCS*, pages 170–184. Springer-Verlag, 2007.

[128] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. In *RE '06*, pages 16–25, Washington, DC, USA, 2006. IEEE Computer Society.

[129] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[130] Reinhard Wolfinger, Stephan Reiter, Deepak Dhungana, Paul Grunbacher, and Herbert Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *ICCBSS '08*, pages 21–30, Washington, DC, USA, 2008. IEEE Computer Society.

[131] Ji Zhang and Betty H.C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006. Architecting Dependable Systems.

[132] Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. Change impact analysis for aspectj programs. In *ICSM*, pages 87–96, 2008.

[133] Chunying Zhao and et al. Pattern-based design evolution using graph transformation. *J. Vis. Lang. Comput.*, 18(4):378–398, 2007.

[134] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

Samenvatting

Wegens vraag uit de markt en veranderingen in de omgeving, dienen software- systemen te evolueren. Echter, de grootte en de complexiteit van de huidige software-systemen maken het tijdrovend om veranderingen door te voeren. Bij onze samenwerking met de industrie hebben wij geconstateerd dat de ontwikkelaars veel tijd besteden aan de volgende evolutie-problemen: het ontwerpen van tijdens runtime herconfigureerbare software, het in acht nemen van beperkingen van het software-ontwerp tijdens het werken aan evolutie, en het hergebruiken van oude oplossingen voor nieuwe evolutie-problemen. Dit proefschrift presenteert drie processen met bijbehorende gereedschappen die de ontwikkelaars/ontwerpers ondersteunen bij deze problemen. Het eerste proces met bijbehorende gereedschappen maakt vroegtijdige verificatie van vereisten aan tijdens runtime herconfigureerbare software mogelijk. Herconfiguratie tijdens runtime wordt gebruikt om software-systemen aan te passen aan de behoeften van de gebruikers en aan de beschikbare hardware. Tijdens runtime herconfigureerbare systemen vereisen speciale aandacht tijdens de ontwerpfase. Met name tijdens evolutie moet men erop bedacht zijn de vereisten van de herconfiguratie van de software niet te schenden. In het algemeen, hoe de software zichzelf herconfigureert moet expliciet gemodelleerd worden in het architectonische model. Het is gebruikelijk dat de verificatie van de vereisten voor de herconfiguratie wordt verricht tijdens de implementatiefase, hetgeen de ontwikkeltijd doet toenemen.

Wij pakken deze problemen aan met een nieuw proces en gereedschappenset om de verificatie van UML-modellen met betrekking tot vereisten voor herconfiguratie tijdens runtime te automatiseren. Bij dit proces worden de UML- modellen omgezet in een op grafen gebaseerd model. De uitvoeringssemantiek van UML wordt gemodelleerd door graaf-herschrijfregels. Met behulp van deze graaf-herschrijfregels en een graaf-productiegereedschap wordt de uitvoering van de UML-modellen gesimuleerd. De simulatie brengt een toestandsruimte voort die alle mogelijke herconfiguraties van de modellen toont. De vereisten voor herconfiguratie tijdens runtime worden uitgedrukt in computational tree logic (CTL) of in een visuele toestandgebaseerde taal (VSL), die wordt omgezet naar CTL. De toestandsruimte wordt doorlopen met een verificatie-algoritme om de toestanden te vinden die aan de CTL-formule vol-

doen. We hebben ook twee mechanismen ontwikkeld om foutmeldingen op te leveren wanneer de verificatie mislukt: 1) gebaseerd op het tracen van de CTL-formule om de plaats waar evaluatie van de formule false oplevert te vinden, 2) gebaseerd op een controle automaat wordt de uitvoeringsreeks van de herconfiguratie opgeleverd (met behulp van VSL) en de simulatie probeert deze uitvoeringsreeks voort te brengen. We hebben experimenten/case studies verricht om de effectiviteit van beide mechanismen te evalueren.

Het tweede proces met bijbehorende gereedschappen werd ontwikkeld voor het met behulp van de computer opsporen van schendingen van statische programma-voorwaarden. Doorgaans hebben software-artefacten statische programma-voorwaarden en aan deze voorwaarden moet voldaan worden bij ieder hergebruik. Bovendien moeten ontwikkelaars voldoen aan de codeer-conventies die in gebruik zijn bij hun organisatie. Omdat in een complex software-systeem aan veel codeer-conventies en programma-voorwaarden de hand moet worden gehouden, is het een omslachtige taak deze allemaal handmatig te controleren. Huidige gereedschappen die ontwikkeld zijn om het controleren op schending van programma-bvoorwaarden te automatiseren gebruiken broncode bevraging en/of uitbreidingen van type-systemen. Een beperking van deze gereedschappen is dat zij werken op abstracte syntax-bomen (ASBs) en geen adequate feedback verschaffen wanneer schending van voorwaarden is geconstateerd. De ASB bevindt zich op een ander niveau van abstractie dan de broncode waarmee de ontwikkelaar werkt, zodat voorwaarden op programma-elementen die zichtbaar zijn in de broncode die de ontwikkelaar gebruikt en waarmee hij bekend is niet geverifierd kunnen worden. We ontwikkelden een modelleertaal, SCML, waarin programma-elementen van de broncode kunnen worden gerepresenteerd. In het voorgestelde proces wordt de broncode omgezet naar SCML modellen. De detectie van schending van voorwaarden wordt gedaan door graaf-transformatieregels, die ook in SCML gemodelleerd worden; de regels detecteren de schending en extraheren informatie uit het SCML-model van de broncode om feedback over de locatie van het probleem op te leveren. Naar deze informatie kan gevraagd worden vanuit een bevragings-mechanisme dat automatisch de graaf doorzoekt. Het proces is toegepast op een industriel software-systeem en op een software-systeem met open broncode.

Het derde proces met bijbehorende gereedschappen voorziet in verificatie met behulp van de computer of een ontwerp-idioom gebruikt kan worden om een verzoek tot verandering te implementeren. Ontwikkelaars neigen ertoe om verzoeken tot evolutie te implementeren met gebruikmaking van software-structuren waarmee zij vertrouwd zijn; deze structuren noemen we ontwerp-idiomen. Het implementeren van ontwerp-idiomen vereist dat de gebruiker een werkschema volgt, een stap-voor-stap beschrijving. Iedere stap van dit werkschema kent invarianten die cruciaal zijn voor de correcte werking van het idioom en moet worden geimplementeerd. Deze invarianten, echter, kennen voorwaarden die vervuld moeten zijn voordat zij

worden geimplementeerd. In de huidige praktijk wordt de toepasbaarheid van een ontwerp-idioom op een wijzigingsverzoek handmatig getest. In ons proces wordt het werkschema voor een veranderings-idioom vastgesteld, en de invarianten van iedere stap worden geextraheerd uit de broncode door deskundigen. Omdat de invarianten uit de broncode worden geextraheerd, kan het zijn dat zij afhangen van programma-elementen die alleen zichtbaar zijn in de broncode. Daarenboven, deze invarianten kunnen zulke programma-elementen ook bevatten. Het SCML-metamodel bevat deze elementen; daarom wordt in dit proces de broncode omgezet naar modellen in SCML. De verificatie van invarianten wordt gedaan aan de hand van deze modellen. Graaf-transformaties worden gebruikt om vast te stellen of de voorwaarden van de invarianten vervuld zijn of niet. Als de voorwaarden vervuld zijn, dan voegen de transformatieregels de broncode die de invarianten implementeert toe. invarianten . Voor een gegeven ontwerp-idioom en gegeven files met broncode wordt het werkschema gesimuleerd. Als alle stappen van het werkschema kunnen worden uitgevoerd, dan zijn alle invarianten van het ontwerp-idioom geinplementeerd in de resulterende SCML-modellen. Daarom worden de modellen weer omgezet in bestanden met broncode. Als, daarentegen, een stap niet kan worden uitgevoerd, dan kan het ontwerp-idioom niet worden toegepast en wordt informatie over de mislukte stap aangeboden. De aanpak is toegepast op een software-systeem met open broncode en een industrieel software-systeem om zijn toepasbaarheid aan te tonen.