

# A Framework for Testing fUML Models

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

**Doctor of Technical Sciences**

within the

**Vienna PhD School of Informatics**

by

**Dipl.-Ing Stefan Mijatov, MSc**

Registration Number 1128521

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel

External reviewers:

Univ.Prof. Dr. Ruth Breu. Univeristät Innsbruck, Austria.

Univ.Prof. Dr. Franz Wotawa. TU Graz, Austria.

Vienna, 26<sup>th</sup> February, 2018

---

Stefan Mijatov

---

Gerti Kappel



# Declaration of Authorship

Dipl.-Ing Stefan Mijatov, MSc  
A-1040 Wien, Karlsplatz 13

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 26<sup>th</sup> February, 2018

---

Stefan Mijatov



# Acknowledgements

I would like to thank professor Gerti Kappel for her guidance and unreserved support during my studies at TU Wien. Furthermore, I would like to thank Philip Langer and especially Tanja Mayerhofer for their help and guidance, while working as part of their team at TU.

Beyond shadow of doubt, without the loving support of my dear wife and my family I would have never achieved as much as I did - I thank you endlessly.

Finally, I dedicate this dissertation to my grandfather, who always inspired the thirst for knowledge and faith in myself.



# Abstract

Software industry is constantly looking for ways to improve the productivity of the software development process, as well as the quality and durability of the developed software product. A significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the implementation domain of a developed solution.

Model Driven Software Engineering (MDSE) is an approach to software development whose aim is the automation of the development process through the specification of models containing domain specific knowledge of the system under development, and transformation of such models into the implementation of the system. Based on the premise that the implementation code is not the main result of the development process, but rather the system knowledge encoded inside the models, starting point in MDSE are the conceptual and implementation independent models of the domain knowledge which are then transformed, according to some formal rules, into implementations on selected target environments.

One of the important issues when using the MDSE approach is that once the implementation artifacts are produced from the models, any existing defects at the model level get transferred to the implementation level, where it is more expensive, in terms of time and effort, to detect and correct them. To improve the development process when using a model driven approach, adequate means for detecting and correcting defects already on the model level are necessary.

One of the most popular modeling languages in MDSE is the Unified Modeling Language (UML), a standard by the Object Management Group (OMG). UML is composed of thirteen diagram types, which can be used for specifying structural and behavioral aspects of a software system. In order to support the execution of models defined with UML, OMG introduced a standard called Semantics of a Foundational Subset for Executable UML Models (fUML), which defines the operational semantics for a subset of UML. Furthermore, a reference implementation of an interpreter that can execute fUML compliant models exists.

The goal of this thesis is to utilize this precise and standardized specification of the semantics and the interpreter of fUML in order to address the lack of testing facilities for fUML models, and thus advance the movement from code-centric to model-centric development, promised by MDSE.





# Contents

<b>Abstract</b>	<b>vii</b>
<b>Contents</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Aim of Work . . . . .	3
1.3 Methodological Approach . . . . .	7
1.4 Structure of Work . . . . .	8
<b>2 Tour of MDSE</b>	<b>11</b>
2.1 Metalevels in the MDA Standard . . . . .	13
2.2 Model Transformations in the MDA Standard . . . . .	14
2.3 Model Execution . . . . .	18
<b>3 Related Work</b>	<b>21</b>
3.1 Testing Approaches in MDSE . . . . .	22
3.2 Formal Method Approaches in MDSE . . . . .	29
3.3 Model Based Testing . . . . .	35
<b>4 fUML Standard</b>	<b>43</b>
4.1 Introduction . . . . .	43
4.2 Modeling Behavior with fUML Activities . . . . .	45
4.3 Execution Semantics of fUML . . . . .	48
<b>5 Testing Framework</b>	<b>53</b>
5.1 ATM Example . . . . .	53
5.2 Testing Framework Requirements . . . . .	57
5.3 Testing Framework Overview . . . . .	59
<b>6 Test Specification Language</b>	<b>63</b>
6.1 Test Language Concepts . . . . .	63
6.2 ATM Example Revisited . . . . .	71
6.3 Test Language Implementation . . . . .	78
	ix

<b>7</b>	<b>Test Interpreter</b>	<b>83</b>
7.1	Overview . . . . .	83
7.2	Trace Model . . . . .	84
7.3	Order Assertions . . . . .	86
7.4	State Assertions . . . . .	94
7.5	OCL Expressions . . . . .	97
7.6	Test Results . . . . .	100
7.7	ATM Example Revisited . . . . .	102
<b>8</b>	<b>Evaluation</b>	<b>107</b>
8.1	User Study . . . . .	107
8.2	Comparison with JUnit Tests . . . . .	119
<b>9</b>	<b>Conclusion and Future Work</b>	<b>121</b>
<b>A</b>	<b>Installing Eclipse Environment and Running the Testing Framework</b>	<b>127</b>
A.1	Installing the Environment . . . . .	127
A.2	Setting up a Project . . . . .	128
A.3	Running the Test Cases . . . . .	128
<b>B</b>	<b>Xtext Implementation of the Test Specification Language</b>	<b>131</b>
	<b>List of Figures</b>	<b>137</b>
	<b>List of Tables</b>	<b>138</b>
	<b>Listings</b>	<b>139</b>
	<b>Bibliography</b>	<b>141</b>
	<b>Curriculum Vitae</b>	<b>149</b>
	Basic Info . . . . .	149
	Educational Background . . . . .	149
	Professional Background . . . . .	149
	Publications . . . . .	150

# Introduction

## 1.1 Motivation

Increased business needs and the advances in computing technologies lead to the development of complex software systems that are meant to operate in highly distributed environments, consist of diverse devices that communicate using a variety of interaction paradigms, can adapt to changes, and are at the same time stable enough to be reliable [FR07].

In order to cope with the increasing complexity of these software systems, advances in the used programming languages, such as concurrency and object-relational mapping, are being constantly made. However, despite of these advances, the gap between the problem domain and the implementation technologies with which the software systems are made, leads to significant decrease of productivity of the software development process and the quality of the final software product [Fra02].

In an effort to decrease this gap, researchers and developers in the area of model driven software engineering (MDSE), are creating modeling languages at a higher level of abstraction than traditional general purpose programming languages.

Models created using higher level modeling languages are described using concepts that are closer to the problem domain, than to the implementation technology used for creating the software systems. The technical details of the implementation environment on which the system will run are abstracted away, thus enabling to more easily reason about the structure and the behavior of the developed system.

Furthermore, using state-of-the-art techniques in MDSE, such as model-to-model and model-to-text transformations, implementation artifacts such as implementation code, database schema, configuration files and other, can be automatically produced from these models.

By automating the development process of the implementation artifacts, a specification of the system under development is separated from its implementation, thus improving the *portability* of the system.

Once a new technical platform is created, an existing specification of a system can be automatically translated into a solution based on the new platform.

Furthermore, making changes to the system at a higher level of abstraction, rather than changing the implementation code, is less error prone and more efficient, as many details of the implementation are hidden from the developer. This leads to the improvement of the *productivity* of the development process, as well as of the *maintainability* of a developed software system.

However, once the implementation artifacts are produced from the models, any existing defects at the model level get transferred to the implementation level, where it is more expensive, in terms of time and effort, to detect and correct them. To improve the development process when using a model driven approach, adequate means for detecting and correcting defects already on the model level are necessary.

In the context of general purpose programming languages several kinds of testing approaches for ensuring the quality of the implementation code, such as unit testing, integration testing and system testing, are adopted. Among these, unit testing is frequently adopted in practice, for ensuring the functional correctness of parts of the software system, called units. In procedural and object oriented programming, a unit is often an individual function or a procedure. Unit tests evaluate assertions concerning the expected output and result of an operation invocation of some part of the software program. Specified tests can be executed continuously during the development life cycle to ensure that the introduction of a new functionality into the system or re-factoring of the existing code has not caused an error somewhere else in the system.

Compared to general purpose programming languages, in MDSE the proper tool support for debugging and testing of modeling languages is often missing. In order to enable testing the functional correctness of models, the corresponding modeling languages must have precisely defined semantics, which would enable their interpretation and execution. Unfortunately, most of the modeling languages in MDSE lack the precise definition of their semantics or contain partially defined semantics, leading to different interpretations by different individuals.

Moreover, most of the existing approaches supporting the validation and verification of models are based on translating these models to some other formalism, such as Petri Nets [Rei85], for which analysis tools already exist. This complicates the validation and verification process by introducing the *forward-translation* of models into some other formalism so that they can be executed, and the *backward-translation* of the validation and verification results to the model level. Both *forward-* and *backward-translation* lead to increased complexity of the validation and verification process, as they introduce an additional layer between the model under test and the specification of the test cases.

Furthermore, the *backward-translation* can lead to loss of information as the results have to be translated back to the model level.

In 2011, the Object Management Group (OMG) has created a standard called *Semantics of a Foundational Subset for Executable UML models* (fUML) [Obj11], specifying the semantics of a subset of the Unified Modeling Language (UML) [Obj15] precisely enough so that the conform models can be directly processed by a machine. Using fUML, it is possible to specify both structural and behavioral aspects of a system under development in a standardized way, and additionally execute the models without the need of a translation to another specification.

However, to support the move from code centric to model centric development of the MDSE approach using the fUML subset, necessary tools for debugging and testing of fUML models are required. A debugger for fUML conformant models has been developed and presented by [MLK12]. However, in order to ensure the quality of fUML conformant models, necessary facilities for testing such models are still missing.

## 1.2 Aim of Work

One of the most popular modeling languages in MDSE is the Unified Modeling Language (UML) [Obj15], a standard by the Object Management Group (OMG). UML is composed of thirteen diagram types, which can be used for specifying structural and behavioral aspects of a software system [HWRK11]. However, the specification of the semantics of UML, which is a prerequisite for executing and testing UML models, is scattered across available documentation and expressed in natural English language, leading to different interpretations of the same specification by different individuals.

In order to address this issue, OMG introduced a standard called *Semantics of a Foundational Subset for Executable UML Models* (fUML) [Obj11], which defines the operational semantics for a subset of UML. This subset is composed of the most relevant part of class diagrams for modeling the structure and activity diagrams for specifying the behavior of the system. Furthermore, a reference implementation of an interpreter that can execute fUML compliant models exists.

The aim of this work is to utilize this precise and standardized specification of the semantics and the interpreter of fUML to address the lack of testing facilities for fUML models. More precisely, the aim is to provide means for validating the functional correctness of fUML models, by testing the modeled behavior specified using fUML activity diagrams. We adopt the concepts of unit testing for empowering users to maintain a high quality of fUML activity diagrams systematically and continuously during the modeling process.

Therefore, we have developed a dedicated test specification language and an interpreter for specifying and executing test cases for fUML models enabling the validation of the correct behavior of fUML activities. Using the test specification language, a modeler can specify assertions on the execution order of the activity nodes, input and output values, and the runtime state of the model.

The test interpreter is based on the extensions of the reference implementation of the fUML virtual machine [MLK12], which is used to execute the activities under test and to obtain execution traces, that are then used for evaluating the assertions defined in the test specification.

The execution trace contains information regarding the *chronological order* of executed activity nodes, the *input and output relationships* between the activity nodes describing which inputs (outputs) were provided to (produced by) which nodes in the executed activity, and the *logical order* describing which execution of nodes enabled the execution of other nodes in the executed activity.

Based on a performed user study (cf. Section 8) there were indications that the developed testing framework may help in ensuring the functional correctness of fUML models, by providing means for detecting and correcting defects at the model level. Furthermore, by having means to specify the test data and the test cases at the model level, the test creation phase of the testing process may be simplified, leading to the increase of its productivity. Additionally, specification of the test data and the test cases (*testware*) at the model level may lead to increase of the maintainability of the testware itself, which was indicated by a comparison of a set of test cases specified with our testing framework on one side, and the JUnit testing framework on the other (cf. Section 8.2).

Having a set of test cases that are separated from the model under test and can be easily repeated when new changes are introduced into the model, enables to support regression testing. Finally, results at the model level can provide information in a more understandable form, as they can be directly integrated into editing tools for better visualization.

### 1.2.1 Challenges

When designing and developing a testing framework for models compliant to the fUML subset, there is a number of challenges that have to be addressed. These challenges come both from the specific nature of the models, as well as the testing process itself.

**Challenge 1: Different levels of abstraction.** UML can be used for sketching and planning a system, as well as for specifying a system precisely enough to generate executable code. Thus, the UML model may be specified at different levels of abstraction. This has to be taken into account in the design of a testing framework for UML models.

For instance, an action within a UML activity may simply be named *Check Application*, whereas no more details are available. In contrast, users may also use low-level actions, such as *Create Object* or *Add Structural Feature Value* for specifying low-level data manipulation. Depending on the level of abstraction, the models under test may vary significantly, thus impacting the specification of the appropriate test cases.

**Challenge 2: Concurrency.** fUML activities provide modeling concepts for specifying concurrent execution flows (e.g., fork nodes). Concurrency in an activity leads to the existence of a potentially large number of possible execution paths of that activity, which

have to be considered in the test evaluation. In particular, checking the correct execution order of the activity nodes has to be evaluated for each possible execution path of the activity under test. Furthermore, a user might be interested only in parts of an execution path, checking only the order of some activity nodes, while ignoring the execution order of rest of the nodes in the path.

**Challenge 3: Test language design.** One important question to address is how to provide means for specifying the inputs needed by the fUML virtual machine for executing an activity under test, as well as means for asserting the correct behavior of activities under test based on the runtime information provided by the fUML virtual machine. Furthermore, it might be useful to specify an initial state, in which the system should be prior to the execution of a specified set of test cases. Decisions regarding these questions influence both the complexity and flexibility of the language, as well as the complexity of evaluating specified test cases by the test language interpreter.

**Challenge 4: Test results design.** Finally, the question how to specify and present useful feedback to the user about the outcome of test executions has to be addressed. In order to provide useful feedback concerning a test result, information regarding the success or failure of each test case has to be presented to the user in a concise and useful format. Based on the presented results, possible causes of a defect and therefore the required corrections shall be more easily inferable by the user.

### 1.2.2 Contributions

The main contributions of this thesis are an executable test specification language and an environment for testing fUML models. The environment is composed of an editor for specifying the test cases in the test specification language, and an interpreter for executing the test cases. With this test specification language and the environment, we aim at establishing the means for specifying and executing test cases precisely and efficiently to support users in maintaining a high quality of fUML models.

Therefore, we leverage the semantics of UML standardized by the OMG in the fUML standard. With providing means for testing UML models on the model level, we aim to foster the promised move from code centric to model centric development and to contribute to a more complete set of tools for model driven development of software using the UML standard. In the following, we summarize the contributions of this thesis.

**Contribution 1: Design of a dedicated test specification language for fUML models.** To tackle the challenges of specifying test input data and test cases on the model level taking multiple abstraction levels and concurrency in fUML models into account (Challenges 1-3 from Section 1.2.1), we have designed a dedicated test specification language for fUML models. In order to provide input to, as well as defining the expected output from an fUML activity under test, it is necessary to specify objects and links comprising the initial or expected states of the system under test, and make them accessible to a set of test cases.

For this purpose, we have designed the concept of a test scenario, as a component of a test suite, which is composed of objects and links comprising a state of the system under test and can be used for specifying the input data for the activity under test, an initial state of the system prior to execution of a test case, as well as the expected output of the execution of the activity under test.

Another important issue is the specification of assertions for evaluating the state of the system during the execution of an activity under test. The execution state of an activity under test is composed of objects and links provided as input or output of the activity nodes within the activity under test. These states describe the structure of the system under test at a certain point in time of the activity execution. Assertions on these execution states may be specified by directly checking the values provided to or from activity nodes of the activity under test, or by specifying complex constraints on the set of objects and links comprising an execution state at the certain point in time.

To address this challenge, each test case in our test language is specified for an activity execution with a defined input as a set of values provided as input to the activity, and is composed of a number of assertions. These assertions can be used for validating the order of activity node executions, as well as for asserting the state of the execution at the certain point in time. Chapter 6 describes in detail the design of the testing language.

**Contribution 2: Development environment enabling to create test cases more efficiently using a dedicated test editor.** In order to improve productivity of the test creation process, we have developed an environment composed of an editor for the test specification language, enabling to specify test cases precisely and easily. This contribution relates directly to the challenge 3 from Section 1.2.1, as it guides the user in the test design process while using our test specification language, and directly contributes to the usability of the language. The editor enables referencing elements of a UML model under test from a test case directly, with support for scoping and code completion. Furthermore, beside standard syntax checking and highlighting, the editor contains additional validation rules which can guide a user in correctly specifying the test input data and the test cases. Details of the implementation of the test specification language are given in Chapter 6.

**Contribution 3: Framework for executing and validating the test cases.** We have developed a framework for executing the test cases created with our test specification language, enabling the validation of the functional correctness of fUML models (Challenges 1-3 from Section 1.2.1). The framework is based on the extensions of the reference implementation of the fUML virtual machine [MLK12].

The activity under test is executed with the test data specified within the test case by the virtual machine. Thereof, the information regarding the chronological and logical order of the executed activity nodes, as well as the execution states composed of objects and links created and modified during the activity execution are captured. Once the activity is executed and the aforementioned execution trace information is recorded, the assertions defined within the tests are evaluated against the trace, and the test results



are recorded and presented to the user. Chapter 7 details the developed framework.

**Contribution 4: Test results model.** We have developed a model of the test results produced by executing the test cases (Challenge 4 from Section 1.2.1). The main component of the model is a test suite result, composed of test case results for each executed test case, containing information about the activity under test, and provided input and output of the activity. Furthermore, a test case result is composed of assertion results, for each assertion in the test case, providing information on the expected and the real value produced by the activity under test (i.e., expected versus actual value produced by an action within the activity, or specified node execution order versus possible node execution order).

This model can be leveraged for building additional testing capabilities, such as the calculation of test metrics, fault localization, and result visualization directly in a model editor. An overview of the test results model and visualization is given in Chapter 7.

### 1.3 Methodological Approach

In this thesis we have applied the design science research approach by Hevner et al. [HMPR04]. Design science creates and evaluates IT artifacts intended to solve identified organizational problems. Hevner et al. define a framework for understanding, executing, and evaluating IT research.

The core of the approach is the development and evaluation of a design artifact. The artifact is designed to solve an identified problem or a need of the environment, thus, assuring the relevance of the research. After the design phase, the artifact is evaluated, and new knowledge is added to the knowledge base.

We outline the way this approach is applied to this thesis, according to the design science research guidelines [HMPR04].

**Design as an artifact.** The result of this research is a testing framework composed of a test specification language and a test interpreter, created with the aim of addressing the lack of testing capabilities for fUML models, based on an OMG standard and the reference implementation of an interpreter enabling the execution of such models.

**Problem relevance.** Tools for validation and verification of models existed long before UML. However, they were constrained to their own domain and proprietary execution semantics, and thus could not be interchanged with other tools and environments.

With the advent of the fUML standard, executable UML models enable the testing and validation process in a more effective and efficient way, by providing means to specify and present the test cases and the test results at the model level.

**Design evaluation.** During the development tests were performed in order to refine and improve the design and implementation of the test specification language and the test interpreter, as well as to ensure their correctness. Also, the ease of use and usefulness of the framework were evaluated by performing a well-structured user study.

**Research contributions.** The contributions of this thesis are the developed test specification language for fUML models, and the environment composed of an editor and an interpreter for specifying and executing the test cases, enabling the effective and efficient testing of fUML models. The contributions were published and reviewed by researchers in the MDSE community and their relevance was confirmed.

**Research rigor.** In the development and evaluation of the artifacts, rigorous and well-structured methods following the design science research guidelines were applied. The artifacts were created to address the lack of testing capabilities for executable UML models based on fUML standard. Foundations of the related disciplines, such as model based testing and static analysis of models in the area of MDSE, were evaluated.

**Design as a search process.** The design and implementation of the developed artifact were improved several times during the research process, by continuously testing the developed artifacts, by taking into account the comments from peer reviews and the performed user study.

**Communication of research.** This thesis and the previous work on which the contributions are based, are published to ensure validity and relevance of the approach, building a new knowledge base for further research. The implementation of the framework is part of a larger project called *Moliz*<sup>1</sup> concerned with model execution, debugging and testing, and is available as an open source project.

### 1.4 Structure of Work

In the following, the structure of the thesis is outlined.

**Chapter 2: Tour of MDSE.** In this chapter an overview of Model Driven Software Engineering (MDSE) is presented. We present a general overview of metallevels in MDA standard, transformations, and model execution.

**Chapter 3: Related Work.** In this chapter, we explain the concepts of validation and verification in software development, and present relevant approaches to validation and verification of software systems at the model level, based on testing and formal analysis. In order to make the survey of the related work more comprehensive, we present approaches to model based testing, where tests are specified at the model level, and used for generation of corresponding tests at the implementation level.

**Chapter 4: fUML Standard.** In this chapter an overview of the fUML standard is given. The testing framework, which is the topic of this thesis, is based on the fUML standard and an extension of a virtual machine capable of interpreting the models specified using fUML. The virtual machine and the extensions are presented in this chapter.

**Chapter 5: Testing Framework.** This chapter provides an overview of the developed testing framework. Furthermore, it introduces an example of an ATM machine with

---

<sup>1</sup><http://www.modelexecution.org/>

which the requirements of the testing language and the testing framework are motivated, which are then implemented and presented in the subsequent chapters.

**Chapter 6: Test Specification Language.** An overview of the test specification language and its features is given in this chapter. Each of the features, such as order assertions, state assertions, test scenarios and others are presented in detail (contributions 1-2 from Section 1.2.1). At the end of the chapter, we describe some details of the Xtext framework used for implementing the test specification language and the associated editor.

**Chapter 7: Test Interpreter.** The test interpreter takes as input the fUML models under test, and the test suites described with the testing language, executes the test suites, and gives the results as output. The details of the implementation of each feature of the test interpreter are given in this chapter (contributions 3-4 from Section 1.2.1).

**Chapter 8: Evaluation.** We have performed a user study with eleven participants in order to evaluate *ease of use* and *usefulness* of the test specification language and the testing framework. Details of the user study, provided material, results and lessons learned are given in this chapter. At the end of the chapter, a comparison of the test cases implemented in the testing language at the model level and JUnit at the code level, is presented. This comparison gives some information regarding the complexity of the test case specification and the performance of the framework.

**Chapter 9: Conclusion and Future Work.** Finally, the contributions of this thesis are summarized and their limitations are discussed. In addition, an outlook on the future work is given.

Parts of this thesis have been published in peer-reviewed conferences and workshops [Ste12, MLMK13, MM14, MMLK15].

An initial idea and a prototype version of our test specification language for fUML activity diagrams was presented in [Ste12]. In [MLMK13] we have presented a more mature version of our testing framework, comprising the test specification language and an interpreter capable of executing test cases for asserting the state of an executed fUML model, as well as the execution order of nodes within an activity under test.

In [MM14] we have presented an overview of challenges in validating the functional correctness of intra- and inter-organizational business process models, and an overview of our testing framework and how it fits into this context.

Finally, in [MMLK15] we have presented our most recent version of the test specification language and interpreter. The newly introduced features of the framework address the concurrency in an activity under test when evaluating the execution order of activity nodes, refinement of temporal and state operators for specifying state assertions, and the use of OCL [Obj12] for specification of time frames and state expressions within the state assertions. Furthermore, we have performed and presented a user study in order to evaluate the ease of use and usefulness of our testing framework. Results of the evaluation and lessons learned were presented.



## Tour of MDSE

Since its beginning software industry was looking for ways to improve the productivity of the software development process, as well as the quality and durability of the developed software product [Fra02]. A significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the implementation domain of a developed solution [FR07].

Main problems and concerns of classic software development approaches, whether its a waterfall, iterative-incremental, agile or some other approach, are productivity, portability, interoperability, maintenance and documentation. As the development technologies and environments are created, companies are forced to migrate so that they can maintain their competitiveness with others.

This migration is forced by two factors: *(i)* the fact that new technologies often solve problems which old technologies could not, or solve it in a more efficient and effective way, and *(ii)* tool vendors eventually stop supporting old technologies once they become obsolete. The situation can be further complicated by the fact that the new versions of the same technology can be incompatible with the old versions. As a consequence of these factors, existing software system is either transferred to the new version of the used technology, or a completely new solution is developed using some new technology.

Software applications are not isolated entities, but rather form distributed information system within an intra-organizational, as well as inter-organizational context. Contrary to old approaches imposing the development of isolated monolith applications, contemporary approaches are based on development of software components that have inter-communication, and which are organized into multi-layered distributed software systems. Each of these components is built using the most appropriate technology, with the mechanisms for communicating with other components within the system.

Documenting a developed software system is usually one of the weak points of every software development project. The reason is usually the lack of time, as priority is always

put on the final software product. Lack of documentation is most noticeable in the maintenance phase, where after some time, even the developers of the software system cannot understand how certain functionality has been realized in the system.

Object orientation, component based development, design patterns, and others present different approaches that were developed over the years in order to improve the productivity of the development process, and to address the aforementioned issues related to the software development process. Model Driven Software Engineering (MDSE) builds on these approaches in order to further advance the way software is developed. Primary concern of MDSE is reducing the gap between problem and software implementation domains through the use of technologies that support systematic transformations of problem-level abstractions to software implementations [FR07].

MDSE is an approach to software development whose aim is automation of the development process through specification of models containing domain specific knowledge of the system under development, and transformation of such models into the implementation of the system. Main idea of this approach is the separation of domain knowledge of a software system from the implementation technology used to realize it. MDSE is based on the premise that the implementation code is not the main result of the development process, but rather the system knowledge encoded inside the models. In the approach, starting points are the conceptual and implementation independent models of the domain knowledge which are then transformed, according to some formal rules, into selected implementation environment.

The MDSE approach led to creation of the standard for software architectures called *Model Driven Architecture*<sup>1</sup> (MDA) defined by the OMG. MDA defines models as key artifacts of the development process.

Rothenberg [RWLN89] defines modeling as a cost-effective use of something in place of reality, which is simpler, safer and cheaper to use than reality for a particular purpose. According to Rothenberg, a model is an abstraction of reality allowing us to deal with a real-world problem in a simplified manner, avoiding the unnecessary complexity, danger and irreversibility of reality. There are many practical usages of models, such as statistical, meteorological, biological, ecological, and other. According to Bézivin [B05], computer science may be described as the science of building models of software systems.

Based on the concepts with which a model of a software system is described, and their dependency to the implementation platform in which the system is realized, MDA standard defines three types of models.

- *Computation Independent Model* (CIM) describes a system independently from the computing platform, i.e., any execution platform in which the system can be implemented.

---

<sup>1</sup><http://www.omg.org/mda/>

- *Platform Independent Model* (PIM) describes a system taking the computing platform into account, but independently of the concrete technology in which the system will be implemented.
- *Platform Specific Model* (PSM) describes a system for a concrete implementation technology in which the system is implemented.

By leveraging model transformations, as described in Section 2.2, PSMs can be automatically generated from appropriate PIMs, and thereof code and other implementation artifacts can be generated from PSMs.

## 2.1 Metalevels in the MDA Standard

OMG, an industry driven organization that develops and maintains standards for developing complex distributed software systems, launched the Model Driven Architecture (MDA) as a framework of MDSE standards in 2001 [FR07]. MDA defines four main levels at which models can be specified. These levels are, according to the MDA standard, named from  $M_0$  to  $M_3$ . Each model from a level  $M_i$  defines concepts of a model specified at the level  $M_{i-1}$ , while at the same time its concepts are defined by a model at the level  $M_{i+1}$ .

**$M_0$  level** is the lowest modeling level of the MDA standard. At this level concepts defining concrete objects of the real world are specified. For instance, at this level a concrete information system at runtime can be specified.

**$M_1$  level** constitutes modeling level at which models of a concrete software system are specified. The concepts specified at this level describe concepts at the level  $M_0$ , called instances of the  $M_1$  concepts. As previously described, models of a system at this level can be CIM, PIM or PSM. Each one of these models could model the same system, therefore they are all specified at the same level. By using model transformations, these models can be generated from each other.

**$M_2$  level** is the modeling level at which concepts of a modeling language are defined. Models specified at this level are called metamodels of models defined at the level  $M_1$ , while the models at the level  $M_1$  are called instances of the metamodels at the level  $M_2$ . MDA standard defines a number of modeling languages at the level  $M_2$ , among which the most popular is the Unified Modeling Language (UML) [Obj15]. UML can be used to specify both structural and behavioral aspects of a software system, and is composed of thirteen diagram types.

One of the structural diagram types of UML is the class diagram for modeling concepts and relationships between them in a concrete software system. Class represents a type of a certain object that can exist in a software system. Therefore a concept specified at the level  $M_1$  can be defined as an instance of the UML class concept defined at the level  $M_2$ .

**M<sub>3</sub> level** presents the highest modeling level of the MDA standard at which languages for defining modeling languages at the M<sub>2</sub> level are specified. These languages are called metamodeling languages, and represent metametamodels of metamodels at M<sub>2</sub> level. The MDA standard defines exactly one metamodeling language at the level M<sub>3</sub> called Meta Object Facility (MOF) [Obj14].

An example depicting the metamodeling levels of the MDA standard is presented in Figure 2.1.

As can be seen from the Figure 2.1, at the highest modeling level M<sub>3</sub> of the MDA standard is the MOF metamodeling language. This language contains concepts used for describing modeling languages at the lower level M<sub>2</sub>, such as UML. The concepts of UML at the M<sub>2</sub> level present instances of the concepts at the higher level M<sub>3</sub>. For example, in Figure 2.1 *UML::Generalization* is an instance of the *MOF::Class* concept.

UML contains concepts for describing structural and behavioral models of a software system at the lower level M<sub>1</sub>. For example, in Figure 2.1 the *System::Vehicle* at the level M<sub>1</sub> is an instance of the *UML::Class* at the level M<sub>2</sub>.

Finally, at the level M<sub>0</sub> the objects existing during runtime in a modeled system are presented. For example, an object of the class *System::Car* with a value of the *name* attribute set to *BMW X5 M* is presented at the level M<sub>0</sub>.

## 2.2 Model Transformations in the MDA Standard

Traditionally transformations from one model to another, as well as from a model to the code of a modeled system, or other artifacts such as documentation, used to be manual. Prior to the MDA, there were many case tools which were able to partially generate implementation code from some proprietary PSM, however still there was a need for coding phase to make the generated code complete. Furthermore, such tools were not interoperable, which made it infeasible to transfer a model from one environment to another, or reuse models in different contexts.

MDA focuses developer on creating PIMs. PSMs are generated automatically by applying model-to-model transformations. An effort is needed to develop certain transformation from one metamodel to another, however once specified model-to-model transformation can be applied to infinite number of different PIMs describing different systems. By focusing the developer to specification of PIMs, developers spend more time on creating business processes implemented through a software system, and thus the time between design phase and implementation is shortened, which leads to productivity increase [Fra02].

For once defined PIM it is sufficient to create a transformation into a PSM, leading to increase in portability. If several transformations have been defined for the same PIM into several different PSMs, transformation rules between PSMs could be thereof deduced, which could lead to further increase of portability.



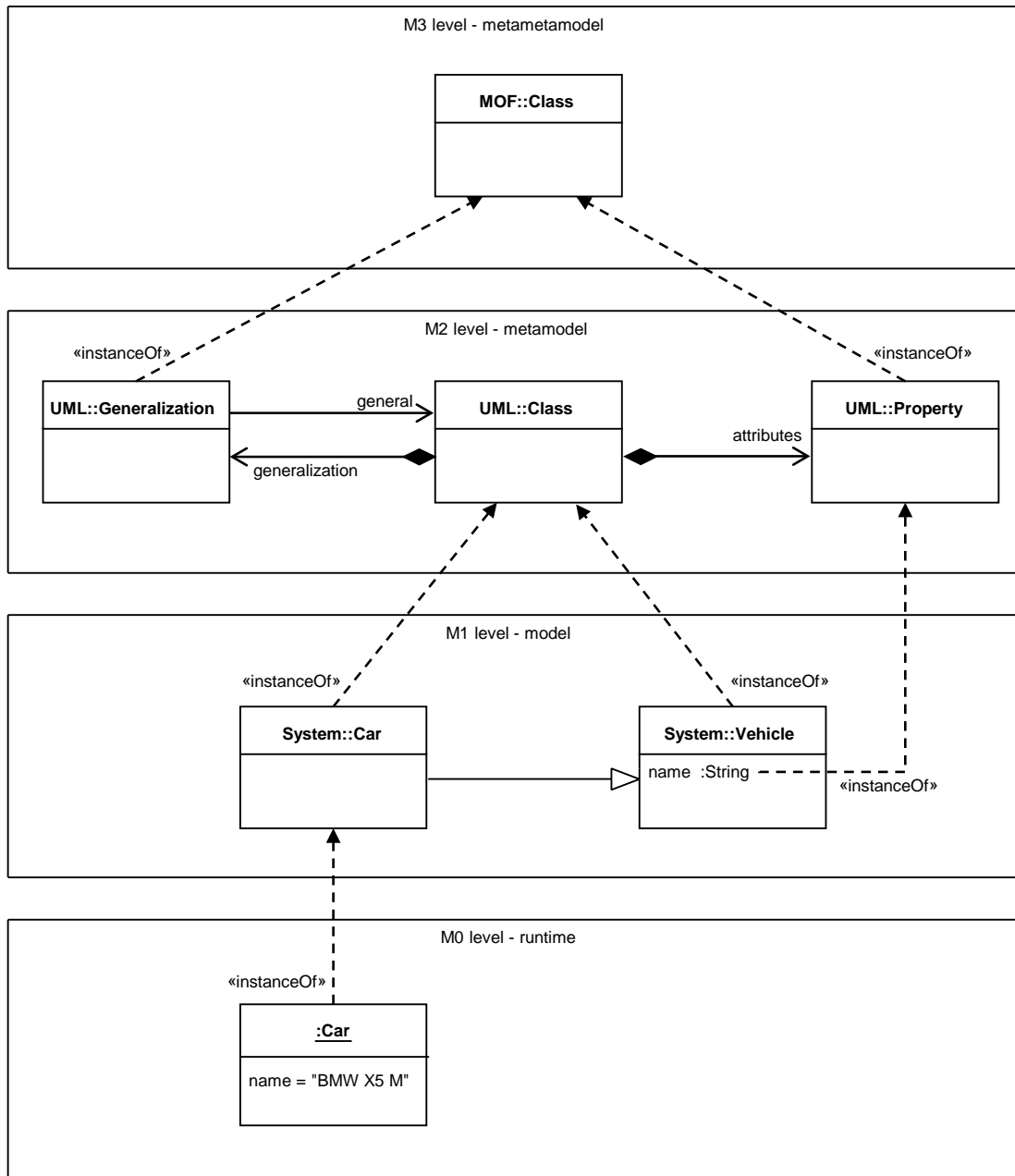


Figure 2.1: Metamodeling levels of the MDA standard, modified version of a figure taken from [Obj15]

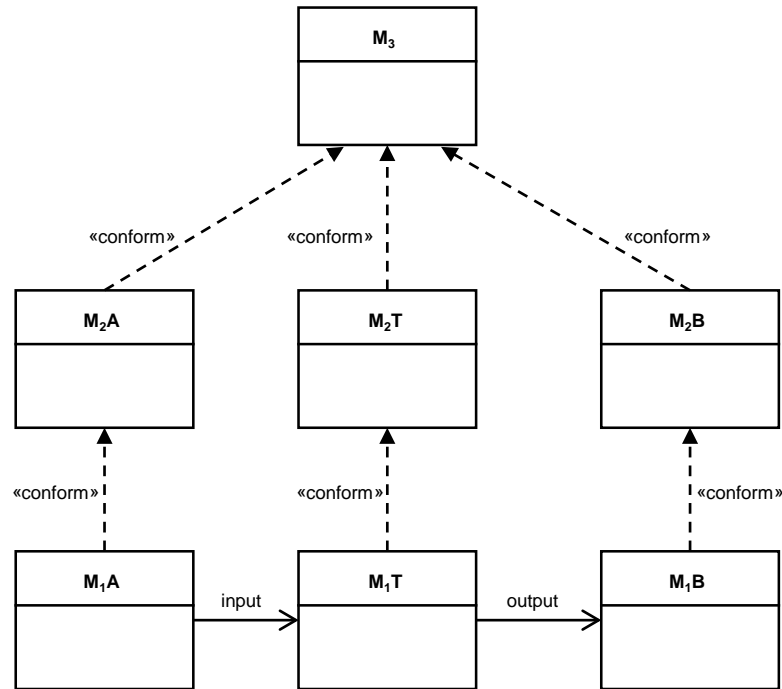


Figure 2.2: Model transformations in the MDA standard, modified version of a figure taken from [ATL06]

In MDSE, the aim of transformations is to provide means for specifying a method of producing certain number of target models from certain number of source models. Transformations should enable a developer to define a way to navigate through elements of a source model, as well as matching the elements of a source model to the elements of a target model. Formally, simple transformation should define a way of producing a model  $M_1B$  which conforms to the metamodel  $M_2B$ , from a model  $M_1A$  which conforms to the metamodel  $M_2A$ . This process is presented in Figure 2.2.

Transformation is defined by the transformation model  $M_1T$  which conforms to the metamodel  $M_2T$  (i.e., the transformation language). This metamodel, as well as metamodels  $M_2A$  and  $M_2B$ , conform to the metametamodel  $M_3$ .

Levi et al. [LAD<sup>+</sup>14] developed a categorization of model transformations, for distinguishing different model transformations according to their properties and goals. Each category corresponds to a *transformation intent* describing the purpose and goals to be achieved by execution of a transformation belonging to the category. Each transformation intent in the categorization catalog is documented by its name, a short description of the general idea behind the intent, and the context specifying scenarios in which a transformation with the described intent can be used.

According to authors of the categorization, there are several main categories of model transformations, some of which are briefly mentioned here.

First category are so called **refinement transformations** which produce output models containing more details than the input models. A refinement transformation, for instance, produces an output platform specific model from an input platform independent model. Transformations for generating source code or serialized form of an input model fall under this category.

Second category are **abstraction transformations**, which are inverse form of the refinement transformations. Their goal is removal of unnecessary details of the input model in order to produce a more generalized output model. Example of an abstraction transformation is one which is used for specifying a query over an input model to produce an output model representing a subset of the input model.

Transformations which produce output models satisfying a set of constraints from an input model fall into the category called **constraint satisfaction transformations**. Such transformations can be used for generating correct instances of a given metamodel, or a set of instances that satisfy certain constraints.

Another category of transformations are **translation transformations**. These transformations are used to produce an output model defined in a different modeling language from the one used for defining the input model.

One example of transformations that fall into this category are so called migration transformations. Migration transformations produce a model of the system defined in one programming language or framework, from a model conforming to another programming language or framework, facilitating the automated migration of a software system from one platform to another.

**Semantic definition transformations** are used to define the semantics of a modeling language. There are two subgroups of semantic definition transformations.

First subgroup are so called *translational semantic transformations*, which produce output models in a modeling language with well defined and known semantics, from an input model defined with a language that has partial or no definition of semantics. In this way, for instance, validation and verification can be applied to the input models by analyzing their analog output models, and converting the results back to the level of input models.

Another subgroup in this category are *simulation transformations*, which apply modifications to the input model producing different states of the modeled system, therefore simulating the system's execution in order to, for instance, facilitate validation.

One issue in MDSE, where models represent main artifacts of the development process from which implementation code is produced by applying model transformations, is that once a transformation is executed, any error created at the model level gets transferred to the code level, where it is much harder to detect and correct it. Being able to execute models created with a modeling language can facilitate creation of tools for validation and verification of such models, further enabling the move to model centric development.

In this respect, transformations from the category of semantic definition transformations are of uttermost importance. In the following subsection, we will describe how model

execution can be achieved through the application of model transformations falling in the category of semantic definition transformations.

### 2.3 Model Execution

With the advancement of MDSE approach, the software development process continuously moves from being code centric to being model centric, where models represent the main artifacts of the development process. As described earlier, the code and other implementation artifacts are automatically produced from these models by applying model-to-model and model-to-text transformations. In this context, the quality of the models, described by both functional and non-functional requirements, becomes essential.

In order to provide means for ensuring the quality of the models, both through static and dynamic analysis, the modeling languages with which these models are described have to be executable. In order to enable executability of models, behavioral semantics of modeling languages used must be specified precisely enough to be processable by a machine.

There are many approaches for specifying behavioral semantics of modeling languages in MDSE area, and they can be categorized into two main groups, namely the *translational semantics* approach and the *operational semantics* approach [CCGT09, KT08]. These two categories directly correspond to translational semantic transformations and simulation transformations defined as semantics definition transformations in [LAD<sup>+</sup>14] as introduced in Section 2.2.

These two groups of approaches can be also understood as building compilers (translational semantics) or interpreters (operational semantics) for modeling languages. In the following, we will describe these approaches and present some examples for each.

**Translational approach.** This approach is based on mapping concepts of an input modeling language (called *source language*) to respective concepts of another language (called *target language*), for which precisely specified semantics is available. Models described with the input modeling language can be translated into the target language, and can thereof be executed within the context of the target language.

One example of a translational approach is a DSL debugging framework presented by Wu et al. [Wu06]. In this approach, Wu reuses an existing general purpose language (GPL) debugger to build a DSL debugging framework. A mapping of the correspondences between the DSL and the generated GPL code is produced, composed of a source code mapping, debugging methods mapping, and debugging results mapping.

Another example of a translational approach where the semantics of UML activity diagrams are defined by their translation into Petri Nets is presented by Störrle et al. [Stö04a, Stö04c, Stö04b, Stö05, SH05]. Concepts of UML activity diagrams, such as action node, decision node, control and data flow edges, etc. are mapped into Petri Net places and transitions, and can be thereof executed with existing tools in order to facilitate validation and verification tasks.

The advantage of the translational approach for defining the semantics of a modeling language is the ability to reuse the existing tools for execution of the target language models. Once the mapping from the source language to the target language is performed, the results of verification and validation tasks performed on the models of the target language can be translated back to the models of the source language, providing the same facilities for a source language which exists for the target language.

The main drawback of this approach is the necessity of creation and maintenance of transformations between the source and the target language, as well as the execution results back to the source language. Furthermore, this additional level of indirection affects both the applicability of the approach due to correspondence between concepts of the source and the target language, as well as the complexity of the developed tools.

**Operational semantics approach.** Another approach to defining semantics of a modeling language is by specifying execution steps for a modeling language, in form of rules, which when applied to a conforming model change its structure in a predefined way. Thereof, an *interpreter* of a modeling language would apply the rules to a model, transitioning the modeled system, from one runtime state to another. While runtime states of an executable model can be realized using metamodeling techniques, the execution of the steps of transforming the runtime states from one to another can be done in three ways, namely by means of a general purpose programming language, integration of an action language as part of the modeling language, or by use of model transformation languages for accessing and modifying the runtime states of a modeled system [May14].

Wachsmuth et al. [Wac08] present an approach where an interpreter is developed as a transition system by using the modeling means standardized by OMG. Metamodels are used to model configuration sets (runtime states of a system), and model transformations are used to specify transition relations. A configuration represents the current state of the system being executed, and is composed of objects, their attribute values and links that exist between them. Transition relation is specified by a set of transformation rules which are applied to the current configuration to obtain the next one.

Each of the approaches for specifying the semantics of modeling languages have their advantages and disadvantages, however overall the advantage of the operational semantics approach compared to the translational semantics approach is that there is no level of indirection between source and target modeling language, reducing the complexity of developing and maintaining the execution, debugging and testing tools. However, for each new language, a new set of tools has to be developed.



## Related Work

In Model Driven Software Engineering (MDSE) main artifacts of the software development process are the models created using modeling languages that are at the higher level of abstraction than existing general purpose programming languages. By leveraging model transformations, the implementation artifacts (such as source code, configuration files, and other) are automatically or semi-automatically generated from specified models, thus improving the productivity of the software development process.

Furthermore, with advancement of model execution, models are not only used as higher-level specifications of the software system from which the implementation of a system is generated, but also become primary artifacts of the software development process. As models can be directly interpreted by a machine, they become formal enough to be considered as primary development products, and could potentially take over as implementation artifacts. In order to support such move from code centric to model centric development, promised by MDSE, advanced development methods and tools are necessary at the model level.

This is especially true for ensuring the quality of the produced software already at the model level, as any defects not captured in the models get transferred to the implementation level, where it is more expensive, in terms of time and effort, to detect and correct them. In software development, quality of the produced software is ensured by the validation and verification process.

Software verification and validation is concerned with both ensuring that the product was built according to the requirements and design specification, as well as ensuring that the product meets the user requirements, and that the specification of the requirements was correct in the first place.

In MDSE, validation and verification of the developed models is commonly realized by applying formal methods. In this context, the validation and verification process is composed of a translation of specified models into a certain formalism (e.g., *first-order*

*logic*), and subsequently applying existing tools for checking general correctness properties, such as e.g., absence of deadlocks, and checking that no invalid system states are produced by system operations.

As the development process becomes more model centric through the use of executable models, model testing will increase in popularity as means for verification and validation in MDSE. A test case defines input to the system under test and assertions on the expected output from the execution. Testing provides means for establishing a certain level of confidence that the selected set of functional or non-functional requirements are met, however it cannot prove the absence of defects in the system as a whole [Som06].

Beside testing new or modified functionality of a software system, existing functionality of the system has to be tested on a regular basis to ensure that any regression in the system has not been introduced by additions of new parts. Regression testing, as an important technique in this respect, is concerned with re-running a selected set of tests to ensure that no defects have been introduced into the already existing functionality of the system by addition of some new functionality.

Testing in MDSE can be applied in two ways. Models can be executed with provided test input data, and the output of their execution can be evaluated against a specified expected output. The purpose here is to test the quality of the models themselves.

Another approach to testing in MDSE is the use of models to test the implementation of a modeled system, i.e., the source code. In this approach, test cases at the implementation level are generated from models that specify the intended behavior of the system. Thereby, the test case generation is driven by predefined test objectives. This approach is referred to as *model based testing*.

In Section 3.1 and Section 3.2, we survey existing approaches of ensuring the quality of models based on testing and formal methods, respectively. In order to make the survey of the state of the art more complete, in Section 3.3 we survey some of the existing approaches to model based testing.

## 3.1 Testing Approaches in MDSE

Testing is an important phase of the development process of a software system. It is commonly used for ensuring that the user requirements implemented as part of the system's functionality are fulfilled. There are two general types of user requirements: functional requirements specifying what the system should do, and non-functional requirements specifying the level of reliability, safety and performance of the system during its use (e.g., performance described in terms of resource usage such as memory, CPU time, network load, etc.).

The functional testing process, which is the focus of this work, is a dynamic process performed by executing the system under test with a given set of inputs, and asserting the generated output of the system against the expected output defined by the test



designer. The tests can be used not only to show that the system implements the correct functionality for a specified set of inputs, but it can be also be used repeatedly over the lifetime of the system, to ensure that modification of existing functionality or addition of a new one has not introduced any new defects into the system (commonly referred to as regression testing).

In this subsection we will present several approaches to functional testing of behavioral models of software systems, focusing on models specified using the fUML standard. There has been substantial research done on functional testing of UML based models outside of the scope of fUML. To make the overview of related work more comprehensive we also give an overview of several such works.

For UML 2 activities and actions, Crane and Dingel [CD08] present the ACTi interpreter, which offers several dynamic analysis capabilities, such as reachability and deadlock analysis, as well as assertions on objects during the execution of activities. The *ACTi* interpreter is based on the *System Model* formalization [BVCR06, BVCR07, Tea02]. With the System Model formalization, the meaning of a model M is defined through the set of all possible instances of M, which satisfy all defined constraints of M. In this formalization, static aspects of a modeled system are defined using the bottom layer of UML 2.0 structural foundation, and the constraints of behavioral models are captured by state machines.

The ACTi interpreter was developed using the Java programming language, and supports the interpretation of UML activities. The process of interpreting an activity is composed of several steps. In the first step, the user has to provide a file defining the structure of the system in a form of a class diagram, and an optional file with the specification of objects existing in the initial state prior to activity execution. Furthermore, the user provides a separate file defining the activity to be executed. In the next step, the interpreter initializes the state of the system using the provided files containing the system structure specification and the optional initial system state specification.

Following this step, the interpreter loads the file with the activity specification, and generates a graph representation of the activity following the System Model formalism. With this formalism, the state of the executed activity is encoded and modified to simulate the behavior of the activity. Execution steps can be performed in two different modes, namely random and guided.

The random execution mode provides the possibility to use randomization for all non-deterministic choices in order to obtain different execution paths of the activity being executed. On the other hand, the guided execution mode provides the capability to suggest a path to the interpreter with the manual input, enforcing the order of node executions of the given activity.

Once the activity is executed, a trace of the performed execution is generated by the interpreter. This execution trace contains information regarding the order of node executions constituting the executed path(s) of the activity, as well as information on the evolution of system state performed by the activity execution.

Beside the capability to execute an activity and produce the execution trace with information detailing such an execution, the ACTi interpreter additionally provides several analysis capabilities. More precisely, two main kinds of analysis are possible: performing a path analysis and performing assertions on the system's state after execution.

In the path analysis, the user can specify and validate several different properties of each executed path, such as *desirable nodes* specifying that any path not containing the specified nodes should be considered invalid, *undesirable nodes* which is the opposite of the previous property, *mutual exclusivity* specifying pair of nodes which should not appear on a path in order for it to be considered valid, *precedence* defining a mandatory order of pairs of nodes on any possible path, *times executed* specifying that a certain node should be encountered a predefined number of times in any possible path.

When it comes to path analysis, the advantage of our testing framework compared to ACTi interpreter, is on one hand that it is not necessary to execute each possible path of the activity in order to perform the path analysis. Using the logical dependencies of nodes recorded in the execution trace, our testing interpreter is capable to compute and perform validations on each path without actually executing it, as described in Chapter 7. For instance, the ACTi interpreter requires to either guide the interpreter to execute the activity in a predefined way, or to use randomization and then manually instruct the interpreter to execute the activity a number of times and perform the path analysis.

On the other hand, the disadvantage of our testing framework compared to the ACTi interpreter when it comes to path analysis is the lack of advanced operators for expressing desired paths, in particular, expressing mutually exclusive nodes, and the number of times a specific node should appear in a path. However, other mentioned path analysis of the ACTi interpreter are fully supported by the order assertions of our testing framework described in Chapter 6. The aforementioned disadvantages represent potential extensions for our testing framework, and could be included as future work.

The second kind of analysis that can be performed with ACTi interpreter is the assertion of the state of a modeled system. Assertions are possible only on the properties of objects of the state after the complete execution of the activity. As opposed to this, our testing framework is capable of specifying state assertions against not only properties of objects, but rather complete system state at a given point in time of the activity execution, using the constructs of the test specification language, as well as the well known OCL expression language (cf. Chapters 6 and 7). However, the current version of our testing framework still doesn't support analysis of effects of concurrency in an activity on a state assertion outcome, but rather validates the state assertions against a single executed path.

Beside these two kinds of analysis, the ACTi interpreter supports two additional ones. The first one considers any leftover control tokens in an executed activity, possibly indicating a deadlock. If such a situation occurs, the interpreter warns the user about the situation, and it is up to the user to decide whether the situation represent an invalid state (i.e., deadlock). At the moment, our testing framework does not support this kind

of analysis, representing potential future work.

Another analysis capability of ACTi is static verification of abstract syntax correctness of provided activity models. ACTi is capable to indicate state constraint violations, such as an incoming control flow into an initial node, or an outgoing control flow from a final node. As our testing framework is used for standard fUML models, existing UML model editors can be used for detecting such abstract syntax violations.

Another characteristic of the ACTi interpreter that has to be mentioned is that in comparison to our testing framework, ACTi is not compliant to fUML. In particular, it does not support the action language of fUML but a proprietary action language.

Another approach for testing fUML based models has been presented by Lazăr et al. [LLP<sup>+</sup>10]. In this work, the authors have proposed a tool chain aimed at building and testing executable UML models based on the fUML standard. This tool chain consists of tools for the creation, execution, and testing of the executable fUML models, accompanied by a code generation tool which enables generation of implementation code in general purpose programming languages such as Java and C++, from the created and tested models in previous phases.

In the creation phase, the structural and behavioral models complying with the fUML standard are specified. On one hand, for creating the structural models represented by fUML class diagrams, a standard editor provided by EMF platform is used [SBPM08]. On the other hand, for creating the behavioral models the authors have developed their own action semantics language and an accompanying editor, for specifying the behavioral models complying with the fUML standard. The action language is proposed and presented in an earlier work by the same group [LLP<sup>+</sup>09].

For the execution phase in the tool chain, the authors propose to use the standard reference implementation of the fUML virtual machine developed by Model Driven Solutions<sup>1</sup>. The authors provided the integration of this virtual machine implementation with the Eclipse workbench, so that the specified structural and behavioral models created in the previous phase can be directly provided to the virtual machine. The execution is realized by either passing the activities created with their action language, along with required activity parameter values, or by writing the test activities with their action language and executing them as tests.

Finally, in the generation phase, implementation code can be generated from the specified models. For this purpose, the authors have developed a set of templates specifically designed for activities created using their action language. These templates can be used to generate implementation code in a general purpose programming language such as Java.

The approach to testing fUML models proposed in this work is based on the creation of additional *test* activities designed for testing activities specifying behavior of the modeled

---

<sup>1</sup><http://portal.modeldriven.org/>

system. While this approach enables modelers to express tests on the expected output of activities, asserting the execution order of activity nodes, as well as the evolution of the system's state during execution is not possible.

In a later work done by the same group of authors [CML13], an fUML virtual machine based on the K-framework [Rc10] is proposed. In this work, the authors propose to develop a complete virtual machine for fUML models featuring debugging and testing capabilities. The K-framework is a rule-based rewriting system, where the state of the executed model at some point in time is represented by a configuration composed of a set of actions comprising an activity in the model, and a set of values passed into each of the parameters of those actions. By applying a set of rewriting rules to the initial configuration, the execution of a model is performed, and finally the output pins of each action are mapped to a set of values. The proposed work seems to be in a very early stage of development, and so far no evidence about the existence of an implementation of the proposed debugging and testing tools based on this framework could be found.

One interesting approach for testing of dynamic properties of models based on the UML standard has been presented by Hilken et al. [HHG14]. In this approach, two types of models are defined: a so called *application model*, describing the static and dynamic aspects of a system, and the *filmstrip model*, defining an instance of a complete execution scenario.

The application model is defined by using UML class diagrams for specifying the static aspects, and is further enriched with OCL operation contracts for each of the defined class operations, specifying the dynamic aspects of the system. The OCL operation contracts are specified in terms of OCL pre and post conditions of defined UML class operations.

The filmstrip model is defined as a single object diagram, composed of several object diagrams specifying each system state after an operation execution, and a set of operation calls between these states describing the state transitions. For each object created or modified by an operation execution, an additional instance is created in a corresponding state, recording the state change in terms of object attributes and link end values. Due to the existence of temporal relations between system states, the filmstrip model can be used to analyze dynamic properties of the system, in terms of object attribute values and links that exist between them representing the system's state and state changes during execution.

Similarly to our approach, in this work the static aspects of the system are described in terms of UML class diagrams. Furthermore, an execution of a model is represented by a sequence of UML object diagrams, similarly to our execution trace metamodel (cf. Chapter 7). However, the specification of the behavior is realized in a declarative form, comprised of OCL pre and post conditions of defined UML class operations, which have to be transformed into another formalism to simulate the execution.

In their earlier work [GBR07, GBR05], the authors have presented an approach for dynamic construction of system execution states from a UML model, by transforming the

defined UML class diagrams and OCL constraints into relational logic, which is in turn interpreted and found results are transformed back to the model level. The approach is implemented as part of the UML based specification environment (USE) tool.

USE allows to validate UML and OCL models by constructing snapshots representing system states at a particular point in time with objects, attribute values, and links. These snapshots are represented as object diagrams in the tool. Class invariants and pre and post conditions of operations are specified using OCL [Obj12], which can then be validated on snapshots of the system states. The tool contains a language called *A Snapshot Sequence Language* (ASSL) which can be used for the construction of snapshots in an automatized way, apart from manually giving commands. Thus, it is possible to specify properties a resulting snapshot has to satisfy.

In the USE tool, it is possible to define two different types of tests: a *test case* and a *validation case*. A *test case* certifies that it is allowed to construct a snapshot fulfilling the defined invariants. A *validation case* is used to validate whether a dynamically loaded invariant is a consequence of the given invariants from the specification of the system.

In USE, the testing process is focused on validating the initial and final states of execution of each operation specifying certain functionality in the system. As opposed to this, our testing framework enables validation of each intermediate state within an activity specifying an operation in the system. Furthermore, it is not possible to check the order of execution of steps within an operation, as the assertions on the execution states are defined only by OCL pre and post conditions of defined operations.

Dinh-Trong et al. [DTKG<sup>+</sup>05, DTGF<sup>+</sup>05] present an approach to testing UML models by translating them into an executable form, and comparing the specified expected behavior of the model under test with the actual one observed during testing. The executable form of the models under test are generated from class and activity diagrams. Test input data is generated from class and interaction diagrams. If the observed behavior differs from the expected one, failures are reported.

To support the approach, the authors have developed a tool called Eclipse Plugin for Testing UML Designs (EPTUD). The tool transforms a UML model under test into a testable and executable form, executes the testable form with the test inputs, and reports failures. In the approach, the operations of the classes from the class diagram are defined by specifying corresponding activity diagrams. Each activity diagram specifies a sequence of actions needed to perform the operation.

To support the action semantics for the activity diagrams, the authors had developed an action language called *Java like Action Language* (JAL). JAL supports several types of actions, such as call operation action, calculation action, create and destroy object action, and several others for manipulating the objects and links within a state of the system under test. JAL can be used to express an activity diagram in a textual format.

The first phase of the testing process consists of providing the UML model under test and a set of test adequacy criteria. In the following phase, a set of test cases satisfying the

test adequacy criteria is generated. A test case is a tuple consisting of three components: a prefix  $P$  representing a sequence of events applied to the system's initial state to create a state from which the testing starts, a sequence of system events  $E$  by which the test case is performed, and an oracle  $O$  defining the expected behavior of the system in the form of a set of tuples composed of an OCL constraint  $c_i$  that should be satisfied in a state created after an event  $e_i$ .

Following the test generation phase, the executable design under test is generated, composed of a static structure representing the runtime configurations of the system under test, and a simulation engine generated from the activity diagrams specified using JAL. The engine decodes system events executing the actions within the activity diagrams, and sends a sequence of actions for updating the structure of the system by modifying the runtime configurations.

In the final phase, a scaffolding is added which actually executes the tests and performs validation. Several different types of checks are performed, such as whether pre and post conditions of operations are satisfied, whether the configuration of the system produced by tested operation satisfy defined constraints in class diagrams, and whether the oracle constraints evaluate to true.

Main disadvantage of this approach, compared to our testing framework, is a need for translation of model under test into an own executable form in order to perform a test evaluation. Furthermore, for defining the action semantics of activities specifying the behavior of the model under test, a proprietary non-standard action language is used.

Pilskalns et al. [PAK<sup>+</sup>07] present an approach for testing UML models composed of class and sequence diagrams. The approach defines an aggregate model that combines both structural and behavioral information and can be used to execute the model for testing purposes. OCL class invariants and pre and post conditions of operations are used to validate the correct behavior of models.

The behavior of a modeled system is represented using sequence diagrams, while the structural information is represented using class diagrams. A Testable Aggregate Model (TAM) is constructed from information contained in the original UML model under test. This model allows to generate and execute tests, and represents an aggregation of the information provided by several views of the system represented by sequence diagrams, class diagrams and OCL constraints. The approach consist of three main steps.

First step is building the TAM from the original UML model. This step consists of constructing a directed graph representing each sequence diagram, a set of class and constraint tuples from class diagrams and OCL constraints, and finally combining the two. A directed graph is created by mapping classifiers and sequence message calls from a sequence diagrams to vertices and edges in the directed graph, respectively. Class and constraint tuple consist of a class name, attributes and operations from the class and its supertypes, and OCL constraints. Finally, as the last phase of this step the directed graphs and tuples are combined together by replacing the vertices with corresponding tuples.

Next step is to determine the input model and generate the test cases. Input model consists of sets of values for attributes of classes from the original model that determine the path of traversing the TAM. First the variables that determine the path of execution of the TAM are selected, then the range of the values to be assigned are determined by applying combinatorial techniques, and finally concrete values from the determined ranges are assigned to the variables.

Finally, last step is the execution of the generated tests. This step consists in traversing the TAM using the test cases generated in the previous step. Each tuple in the graph is visited, and the state and trace information is recorded for each path. The trace information is finally validated against the OCL constraints.

For each executed test, potential faults are recorded, and test results are presented. Two different types of faults can be revealed by the test execution. First type of fault is called *path fault*, discovered in case a certain path in the graph is not traversable or does not exist. Second type of fault, called *OCL fault*, occurs when states recorded in the execution trace violate the defined OCL expressions. For the validation of OCL constraints, the USE tool is applied.

As opposed to our testing framework, this approach concerns testing UML class and sequence diagrams of a model under test, and the execution of the model and recording of an execution trace is achieved by a translation of the model under test into an own proprietary executable form. However, when it comes to what is being tested, it is very similar to our approach. As in our testing framework, path validation concerns detection of non-executable or non-existing paths, which can be achieved with our order assertions for an activity under test (cf. Section 6.1.4). Furthermore, validation of OCL constraints in each execution state is similar to our state assertions (cf. Section 6.1.5).

Another interesting line of work related to model testing is work on temporal OCL (e.g., [BGKS13]). Temporal OCL is an extension of OCL with temporal operators and quantifiers enabling not only the evaluation of OCL expressions on a single state of a system but also on its evolution. Thus, temporal OCL could be used in a similar way as our state assertions for testing purposes. However, our testing framework does not extend OCL with temporal expressions, but rather uses it in its original form and instead provides temporal expressions as part of the test specification language.

A summary of the discussed approaches and their comparison with our testing approach is given in Figure 3.1.

## 3.2 Formal Method Approaches in MDSE

Formal methods represent techniques based on mathematical models and tools for the development and analysis of software system [Hax10]. In specification and design phases, using formal methods, it is possible to increase the level of confidence that the developed product corresponds to the specified requirements. They can help in increasing the number of defects found in earlier stages, such as requirements specification and system

### 3. RELATED WORK

Approach	UML/UML Subset	Execution Order Analysis	State Analysis	Test Input Specification
ACTI Interpreter by Crane and Dingel [CD08]	UML Class and Activity diagrams (proprietary action language)	Includes dedicated operators for specifying properties of the expected execution path of an activity: desirable nodes, undesirable nodes, mutual exclusivity, precedence, times executed	Checking the properties of objects in the final execution state of the model	Activity input and initial system state can be defined.
Execution/Analysis Tool Chain by Lazar et al. [LLP+10, CML13]	UML Class and Activity diagrams (proprietary action language)	No support	Support for asserting output of an activity	Support for specifying input of an activity
Application/Filmstrip Model based on USE Tool by Hiken et al. [HHG14]	UML Class diagrams and OCL operation contracts	No support	Validation of object properties and links through OCL invariants, pre and post conditions applied to execution trace	Initial system state can be specified
Eclipse Plugin for Testing UML Designs by Dinh-Trong et al. [DTKG+05, DTGF+05]	UML Class and Activity/Interaction diagrams (proprietary action language)	No support	OCL constraints that should be satisfied in specified states of execution	Test input data is generated from Class and Interaction diagrams
Testable Aggregate Model by Piskalins et al. [PAK+07]	UML Class and Sequence diagrams	Specified paths not traversable or non existing in the input sequence diagram are found and reported	States which violate predefined OCL constraints are reported	Input values for executing the class / sequence diagrams are generated using combinatorial techniques
Temporal OCL by Bill et al. [BGKS+13]	UML Class diagrams and OCL constraints	No support	States which violate predefined OCL constraints are reported	Initial system state can be specified

Figure 3.1: Summary of presented UML testing approaches



design, rather than in the testing and maintenance. As they are usually more expensive and complex than e.g., testing approaches, formal methods are commonly used in the development of safety, business, and mission critical software, where the defects in the final software product might have high impact (e.g., loss of lives or high loss of value) [Hax10]. There are two main parts of a formal method, namely formal specification and formal verification.

A formal specification in software development is a description of a software system expressed in a formal notation, that is a language that has a precise syntax, and where each concept of that language has defined unique mathematical meaning [Hax10]. In the requirements analysis phase of a software development process, a formal specification of a software system is created, giving a description of the requirements which are to be implemented by the system [Hax10]. Formal specifications are abstract in the sense that they omit unnecessary details of the implementation, precise in the sense that they do not leave room for different interpretation, and allow for formal analysis as they are based on precise mathematical formulas.

The formal verification, on the other hand, is the process of investigating whether a software system is correct according to its specification, by applying mathematics [Hax10]. Software verification had been traditionally done by code inspection and testing. However, in large systems due to a huge number of possible system executions it is only possible to cover a small part of them by applying testing techniques. The advantage of formal verification methods is on one hand, that they consider large number of possible executions, and on the other that defects can be found before the system is implemented. The disadvantage, however is that the formal specifications are at the higher level of abstraction than the software system itself, therefore formal verification is usually used as a supplement to testing [Hax10].

There are two main approaches to formal verification, namely theorem proving and model checking. Theorem proving represents the process of constructing a mathematical proof (i.e., strong mathematical argument), for a mathematical statement to be true. Proofs can be classified into two groups: semi-formal proofs written using a mix of mathematical formulas and natural language, and formal proofs which are completely expressed in a formal mathematical language having a precisely defined syntax and semantics.

A proof of a mathematical statement consists of a sequence of argumentation steps, where each step consists of some premises which yield a number of conclusions. Each step is repeated, where the conclusions of a previous step are used as premises in the current step to yield a new set of conclusions. Steps are repeated until the mathematical statement, i.e., theorem, to be proved is drawn as a conclusion in the last step. There are three main groups of computer based tools that are used in theorem proving: proof checkers which automatically check whether a postulated proof is actually a correct proof of a given theorem, interactive theorem provers which can be used to interactively construct a correct proof, and automated theorem provers which automatically search for a proof of a given theorem.

Romero et al. [RSGVF14] discuss a subset of fUML standard called *base semantics*, covering its formal definition and usage for theorem proving, and show how the standardized formal semantics of fUML can be utilized to perform formal verification through theorem proving. Furthermore, they discuss existing deficiencies of the standard in context of formal verification, which need to be improved. We are not aware of any other work applying theorem proving on UML activity diagrams.

Another approach to formal verification in software development is model checking. Model checking presents an automated approach to verify that a model of a software system specified as a finite state automata satisfies a given set of formally specified system requirements. A finite state automata describes a finite set of system states, through which the system evolves over time during its execution. The requirements which are to be verified are formalized as a set of constraints on how the state of the system is allowed to evolve over time [Hax10].

Once the automata of the system and the requirements to be verified are formally specified, a model checker can be applied to verify the requirements. The model checker performs this process by exhaustively exploring all system states, and checking if the constraints specifying the requirements are satisfied in those states. The requirements specification is typically expressed in a temporal logic language containing concepts for defining how a system state can evolve over time. Temporal logic is an extension of propositional logic with operators that can express relations between properties of a system over time during its execution. There are two main groups of properties which are verified during model checking, namely safety properties and liveness properties. Safety properties express that a system never reaches an undesired state, and liveness properties express that the system will eventually reach a desired state [Hax10].

The advantage of model checking over theorem proving is that it is fully automated and therefore much faster and easier to use. However, when checking large systems, the number of states which are needed to be explored might exceed the available computer resources such as processor speed and size of memory, known as state space explosion. In this case, the theorem proving might have an advantage over model checking. Also, some systems cannot be easily formalized as finite state automata, and in these cases the theorem proving can be easier to apply.

As UML activity diagrams can be easily formalized as finite state automata, there are significant research results on applying model checking for their verification. In this section, we present an overview of the research concerned with the verification of functional requirements in UML activity diagrams, based on model checking.

Daw et al. [DMC15] present an approach and an Eclipse Plugin for the verification of UML activity diagrams against specified requirements. The verification process is composed of the translation of UML activity models into a representation suitable for one of several model checkers, such as UPPAAL, SPIN, and NuSMV, and the subsequent verification of user-defined requirements with a selected model checker. The requirements are formulated as LTL or CTL formulas, and fed into the model checker by the plugin.

A verification result is generated in the form of a text file, showing the satisfiability of the given requirements, or presenting a counter example in case the requirements were violated.

Abdelhalim et al. [AST13] present an approach to verification of fUML models by performing an automatic formalization of fUML models into communicating sequential processes without any interaction with the modeler, in order to isolate the modeler from the formal methods domain. The verification process provides the modeler with a UML sequence diagram that represents the model checking result in the case where an error has been found in the model. The approach also considers the formalization of systems consisting of asynchronous communication between components allowing to check dynamic concurrent behaviors of the systems.

The authors have developed a comprehensive framework that is implemented as a plugin to the MagicDraw tool called *Compass*. The framework uses the Epsilon Framework as a model transformation tool that utilizes the MDSE approach. Furthermore, an optimization approach is included in order to be able to formalize concurrent systems, and at the same time comply with the fUML inter-object communication mechanism. The formalization language used for enabling the execution of the modeled system is CSP [Hoa78]. The system is formalized as a set of processes that are executed and can communicate by means of signals.

The verification process is composed of several phases. In the first phase, the modeler develops the fUML model of the system using the case tool MagicDraw. Behavior of each active class in the system is specified by means of an fUML activity diagram. Following the first phase, the fUML is exported into the XML Metadata Interchange (XMI) format.

In the second phase, the model formalizer component reads the XMI representation of the fUML model and transforms it into a CSP script. The model formalizer uses the Epsilon Framework to perform model-to-model and model-to-text transformation tasks. The generated CSP script contains a process for each active class in the system, as well as a formalization of the inter-object communication mechanism to allow those processes to communicate with each other asynchronously via signals.

An object-to-class mapping table is generated by the model formalizer for traceability between the modeler friendly feedback and the original fUML model. In case a problem is detected during the formalization process, the model formalizer generates the formalization report containing information about the errors in the fUML model which led to the problem (e.g., an fUML activity diagram without a connected initial node cannot be formalized).

In the last phase, the framework checks the generated CSP script for deadlocks. In case a deadlock is present, a counter example is generated that includes a sequence of events that led to the deadlock. The UML Sequence Diagram Generator reads the counter example and visualizes it in the form of a UML sequence diagram. The generated sequence diagram represents the deadlock scenario in a modeler friendly format.

Laurent et al. [LBBG14] present an approach to verifying fUML models covering control and data flows of a process modeled using activity diagrams, taking into account resource and time constraints. The formalization is implemented using the Alloy modeling language, and a graphical tool on top of the implementation is provided. The result of the verification process is displayed on the process model diagram.

The verification process addresses several different properties. Questions regarding control flow, such as whether the process terminates or whether a certain task ever gets executed, are analyzed. Data flow analysis addresses the issue of missing data, that is the attempt to access the data which has been previously deleted. Resource and timing analysis addresses the questions regarding missing resources or whether it is possible to finish the process in a given time whatever the path taken. Beside these global properties which should hold for any given process, it is possible to specify and check so called *business properties* representing specific constraints which should hold for a concrete process (e.g. is the action  $A$  executed if the condition  $C$  is fulfilled).

A prototype implementation in the form of an Eclipse EMF plugin is provided. The prototype assists the modeler by automatically verifying fUML processes in the form of XMI instances. The prototype comes with a library of predefined properties that can be checked, and also allows to add business properties through a graphical interface.

Planas et al. [PCG11] propose a lightweight verification method for fUML models, focusing on the verification of the *strong executability correctness* property of action based operations. If every time an operation is invoked, the set of modifications the operation performs on the system leads the system into a new state which is fully consistent with all defined integrity constraints, the operation is *strongly executable* (SE). In the approach the structural models of the system are described using UML class diagrams and OCL, and operations are defined using the Alf Action Language.

Given an input in form of structural and behavioral models, the method returns an answer that an operation is SE, or otherwise gives a corrective feedback consisting of a set of actions and conditions that should be added in order to make the operation SE. After addition of the actions and conditions from the corrective feedback the operation might not be SE, as some additional constraints might be violated. Thus, the process is recursively repeated until the operation becomes SE.

When analyzing the SE of an operation, all of its execution paths must be taken into account. Thus, the process consists of the following steps. In the first step, all of the paths of the operation are computed. Next, each action in each path is analyzed to check if its execution violates any of the integrity constraints from the structural model. Finally, in the last step a contextual analysis of each potentially violating action is performed to check if other actions or conditions compensate the effect of this potentially violating action to achieve a consistent state of the system. If all potential violating actions can be discarded, it is concluded that the operation is SE.

Eshuis and Wieringa [EW04] present a formalization of workflow models specified as UML activity diagrams for verifying functional requirements. In their approach, activity

diagrams are translated into transition systems, functional requirements are defined as LTL formulas, and these LTL formulas are evaluated on the obtained transition systems using the NuSMV model checker.

Beato et al. [MEBSECdlF05] present a tool called TABU (Tool for the Active Behavior of UML), which provides a formal framework for the verification of UML models. The state of the system is represented by UML class diagrams, and the behavior is represented by UML activity diagrams and state machines. Underneath, the tool uses the model checker SMV [McM92] for the verification process, facilitating an automatic translation of defined UML behavioral models into a representation fitted for use within SMV, and an assistant for formulating the properties to be verified by the SMV model checker.

The properties which can be formulated and verified with the tool fall into two categories. First category are so called occurrence properties, that specify that a certain state or an action is never, always, or at least once reached (i.e., traversed) during model execution. Second category are so called order properties describing in which order certain actions or states are traversed during model execution, such as precedence (e.g., action A before action B) and response (state D responds to state C).

Most of the presented model checking approaches for UML / fUML activity diagrams are focused on the verification of global system properties, such as liveness [LBBG14], the existence of deadlocks in a model [AST13], and strong executability of activities [PCG11]. Therefore, as previously mentioned, they can be used to improve the validation process performed by testing approaches.

Some of the approaches, such as the ones proposed by Laurent et al. [LBBG14] and Beato et al. [MEBSECdlF05] address similar properties as our testing framework, such as the order of node execution in an activity, or state related constraints, much in a way our order and state assertions are used.

For instance, the work by Beato et al. [MEBSECdlF05] addresses properties that are similar to the properties addressed by our testing framework. Much like using our testing framework, it is possible to verify some simple properties related to the execution order of UML activities and nodes within activities. However, the tool doesn't support verification of the system state modified during the execution of an activity diagram.

### 3.3 Model Based Testing

Testing a software system to validate whether the implementation fulfills the specified functional or non-functional requirements is an essential phase of the software development process. Time and effort required to create the test cases, execute them, and assess whether they appropriately cover the functionality of the system under development can be significant.

In MDSE models can be used for several purposes. Complete code or code fragments can be automatically generated from the specified models. Furthermore, out of models,

test cases can be derived manually or automatically. Also, the models themselves can be used to describe test cases and thereof generated test cases at the implementation level of the system under test [Rum03]. Last two approaches to using models for the purpose of testing, are usually referred to as model based testing (MBT).

Model based testing represents a technique in software testing, coming from MDSE area, whose intention is to automate the validation process, thus reducing its initial cost. Model based testing tools automatically generate test cases from models of a software system. The generated tests are executable and include an oracle component which assigns a pass/fail verdict to each test [Utt05].

Model based testing is commonly realized in four phases: *building an abstract model of the system under test*, *validating the model* to ensure there are no defects at the model level, *generating abstract tests from the model*, and *refining abstract tests into concrete executable tests*. The two last phases are usually automatic.

Once these phases are done, the concrete tests can be executed on the system under test in order to detect defects at the code level. According to Utting [Utt05] different approaches in model based testing can be differentiated in accordance with several different aspects.

One such aspect is the *nature of the model* under test. The model from which the tests are generated can contain only the concepts of the system under test, or the concepts of the environment in which the system runs, or more commonly the combination of both.

Another aspect is the *model notation*. Different formal specifications such as Z, VML, and Spec# for modeling the system under test in an MBT approach, but also transition based notations such as statecharts and UML state machines have been used.

Third common aspect is *control of test generation*. One approach to controlling the test generation is to specify a model coverage criteria. Most code based criteria, such as statement and decision coverage, have been adopted in practice.

Another aspect is so called *on-line* or *off-line* test generation. On-line model based testing generates tests in parallel with executing them, while off-line model based testing generates tests as a separate step from their execution. This has some advantages, such as repeatability of the generated tests for the purpose of regression testing.

Finally, one additional important aspect for differentiating approaches to model based testing is *requirements traceability*. It might be necessary to record a trace which relates each specified requirement at the model level with the corresponding generated test at the code level. This traceability can help with validation of specified requirements, as well as assessing the achieved coverage criteria.

Depending on the level of abstraction of the model of the system under test (SUT), model based testing can be considered black-box or gray-box [SL15]. An abstract representation of the SUT can lead to black-box model based testing, while a model of the SUT that includes design information leads to gray-box model based testing.

In model based testing, a typical test consists of a description of *test data*, a *test driver*, and an *oracle* characterizing the expected test result. In model based testing using UML [Obj15], the test data can usually be described by a UML object diagram. The object diagram shows the necessary objects as well as concrete values for their attributes and links, needed for execution of part of a system under test. The test driver can be defined as a simple method call, or modeled using a diagram representing the behaviour of the system under test, e.g., UML sequence diagrams or UML activity Diagrams. The usage of behavioural diagrams for specifying the test driver has the advantage that not only triggering method calls can be described, but it is also possible to model desired object interactions and check object states during the test run [Rum03]. Finally, an oracle can be described through a combination of UML object diagrams and OCL [Obj12] constraints.

Holzer et al. [HJK<sup>+</sup>11] describe an application of FQL (FShell Query Language), a code coverage specification language for ANSI C programs, to UML activity diagrams and state machines, in order to automatically generate a set of abstract test cases on the model level, which are then automatically concertized into test cases at the code level.

In FQL, programs under test are represented by control flow automata where vertices represent concrete statements within the program, and edges represent transitions, i.e., branches of control flow between statements. Using FQL it is possible to specify code coverage which should be achieved by generated test cases. More precisely, it is possible to specify code coverage criteria, such as basic blocks coverage and condition coverage specifying which statements and condition edges in the program should be traversed during execution of a generated test suite.

As UML activity diagrams and state machines are easily interpreted as control flow automata, application of FQL for specifying test coverage for such models seems natural. In order to support such specification using FQL, authors have developed a syntax extension of FQL to support expressing path coverage of UML activity diagrams. The extensions support specifying path coverage using statements involving activity nodes and control flow conditions. It is possible to specify which nodes in an activity should be covered by a test case, as well as which control flow conditions should be evaluated to true during a test execution.

In order to facilitate test generation at the model level, a model under test is translated into a corresponding C program, and the specified code coverage statements at the model level are translated into corresponding ones at the code level. Thereof, the FShell interpreter is used to generate a test suite for the C program representing the input model.

The main drawback of this approach is that in order to facilitate the verification process of an activity diagram it has to be first converted into a C program which increases complexity of the approach and might lead to inconsistencies between the original model and the target representation. Furthermore, the results of test execution has to be translated back to the model level, which might lead to inconsistencies or data loss.

It would be interesting to investigate how this approach could be used to generate test cases at the model level specified using our test specification language, and what additional extensions of the test specification language might be needed to support this. This way, translation of input UML activity diagrams into their C program representation, as well as translation of test case results back to the model level can be omitted, and therefore complexity of the verification process reduced and potentially the confidence of the verification process could be increased.

Another interesting approach to model based testing based on UML activity diagrams is presented by Linzhang et al. [LJX<sup>+</sup>04]. In the approach, a set of test scenarios at the model level is generated for a set of UML activity diagrams, which traverse each path and node previously specified to be covered by the test execution. The approach is supported by a tool which as the first step parses an input UML activity diagram, and derives a set of test scenarios satisfying a set of previously defined path coverage criteria. Thereof, the test scenarios are processed to derive a set of test cases at the code level. The concrete input and output parameters for the system under test are derived from the action sequence and guard values in a test scenario obtained at the model level.

Lei et al. [LWL08] present an approach to testing concurrent Java programs by using UML activity diagrams to generate test cases in order to discover any existing data races and potential deadlocks. The approach is based on tagging each action within a UML activity diagram representing the program under test with corresponding annotation indicating a field of an operation or a class being read and written by the actions during the activity execution.

The approach is composed of several steps. In the first step, based on path analysis performed on the input activity diagram, a random set of test cases is generated. The test case generation algorithm is based on the previously described approach by Linzhang et al. [LJX<sup>+</sup>04]. The paths of the input activity diagram to be executed indicated by the generated test cases, along with the data access annotations of actions within the input activity diagrams, are taken as input for generating the instrumentation code for executing the input activity diagram.

In the final (verification) step, based on the previously obtained execution traces, an analysis is performed to identify any existing data races in the modeled process under test. The verification step is performed based on analyzing the actions which can be executed concurrently in a single process or can be executed by multiple threads during a single run of the program, and which read and write to the same variables as declared by the annotations from the input model. Each executed path from the generated set of test cases with actions which comprise a data race is detected and reported.

As described in Chapter 7, our testing framework supports validation of execution orders of an activity under test taking into account any existing concurrency in the activity. However, the state assertions are currently validated only against a single execution path traversed by the fUML virtual machine. It would be interesting to investigate how the approach by Lei et al. [LWL08] could be applied in our testing framework, in order to



detect any potential paths in which data races could occur. If such a path is identified, it could be possible to evaluate each state assertion from the defined test suite in our testing framework against the path.

A similar approach for generating test cases from UML activity diagrams is presented by Kim et al. [KKBK07]. In this approach a UML activity diagram used for test generation is translated into a so called input output activity diagram, which models each action of the original activity as either an accept event or send signal component. Therefore, the activity is modeled in such a way that only external inputs and outputs of the activity are considered, reducing the state space during test case generation.

Chen et al. [MXX] present an approach to test case generation for Java programs modeled as UML activity diagrams. In this approach, instead of deriving a constrained set of test cases satisfying a predefined coverage criteria, this set is randomly generated from the model, and each generated test is executed at the code level to obtain the execution traces from the program under test. Once the traces are generated, they are compared with the behavior defined in the input UML activity and based on a predefined coverage criterion, a selection of test cases from the previously generated set is performed to achieve the coverage goal.

Another interesting approach for generating test cases from UML activity diagrams is presented by Debasish and Debasis [DD09]. In this approach the test generation process is realized in three steps. In the first step, a model of a process under test represented by a UML activity diagram is augmented with additional test information, required for generating the test cases. This information comprises the indication of which objects are created or modified by the activity under test. This is done by indicating which input and output pins consumed or provided which objects. In the next step, the input diagram is converted into a graph representation. This graph is a directed graph in which every node represents an activity node from the input activity. In this graph each pair of forks and joins is replaced by a node representing a higher level activity node, therefore eliminating the state explosion induced by any concurrent behavior within the activity under test.

Finally the graph generated in the previous step is used as input for generating the test cases for the modeled process. Authors defined a so called *activity path* coverage criterion, which assumes that in each generated path from the activity each existing loop is executed at most two times. A set of test cases is generated, such that each test case from the generated set covers a distinct path from the graph, such that the aforementioned coverage criterion is reached.

The test cases target a set of faults to be detected, such as faults in decision nodes and loops, and synchronization faults. A decision node fault occurs when an unexpected alternative path is taken during the activity execution. A loop fault may occur either in entry or exit condition, resulting in loops being not executed at all, or never ending. Furthermore, an unexpected number of iterations in a loop can be detected. Finally, a synchronization fault occurs when an activity starts execution prior to finishing the

execution of all preceding activities.

The order assertions defined within our testing framework (cf. Chapter 6) can be used for detecting decision node and synchronization faults within an activity under test. Furthermore, our approach supports evaluation of order assertions taking into account concurrency modeled within the activity under test. However, the disadvantage of the current version of the testing framework is the lack of loop fault detection. Therefore, it would be interesting to investigate what are the necessary extensions to the testing framework in order to support loop faults detection.

There is a vast amount of research done in the area of model based testing, concerned with generating test cases from UML models other than activity diagrams. Instead of trying to exhaustively describe every possible model based testing approach based on UML models, the related work presented so far was constrained to only those approaches based on UML activity diagrams. To make the overview more comprehensive, we conclude the related work with shortly describing several approaches outside of the UML activity diagram scope.

For instance, in an approach by Bouquet et al. [BGL<sup>+</sup>07] the authors define a subset of UML2 for model based testing purposes. This subset allows designing formal behavior models of the SUT, which can be automatically interpreted to generate test suites. The structure of the SUT is modeled using UML class diagrams, and the test data for the SUT is modeled using UML object diagrams. For modeling the behavioral aspects of the SUT, UML state diagrams are used. The expected behavior of the class operations, as well as transitions between states within state machines, are formalized using OCL constraints.

An approach where test cases at the model level are generated from UML sequence diagrams is presented by Grigorjevs [Gri11]. This approach is based on automated test case generation for timing details verification of the SUT. As input of the test case generation process a sequence diagram in XMI format, along with the transformation rules, are provided. The transformation uses a UML sequence diagram as a source model, and a UML testing profile [Obj13b] as a destination model.

The test generation process consists of several steps. In the first step, an instance of a UML sequence diagram representing the model of the system under test in XMI format is provided as input for the transformation into the analogue representation of the model in a relational database table. In the second step, the relational table representing the system under test is transformed into another relational table representing the system under test, in a more adequate way for the test case generation. Finally, test data is generated out of the model representation from the previous step, by applying defined transformation rules.

There are other approaches, such as the one by Zech et al. [ZFKB12], where test case generation and selection are based on model changes recorded by a versioning system. This approach provides support for regression testing of arbitrary XMI based model representations, that can be parameterized with different change identifications, impact

analyses, and regression test selection strategies defined with OCL. The approach is based on the model repository MoVE supporting versioning of arbitrary models.



# fUML Standard

## 4.1 Introduction

The Unified Modeling Language (UML) is an object-oriented graphical modeling language, developed and standardized by the Object Management Group (OMG), used to specify, construct, visualize, and document both structural and behavioral aspects of a software system under development [Obj15]. UML is widely accepted standard for modeling software systems.

UML consists of the infrastructure, the superstructure, the object constraint language (OCL) [Obj12], and the diagram interchange standard. Infrastructure forms the basis of the language definition of UML, and therewith of the superstructure. Superstructure represents the actual definition of the UML modeling language. Object constraint language can be used for specifying additional constraints on the models, or querying the models. Finally, specification of layout exchange information for UML diagrams between UML tools is defined by the diagram interchange.

Please note that the latest version of UML, UML 2.5 [Obj15], removes the distinction between the infrastructure and the superstructure, leading to simplification of the standard as the complete specification of the language is given in a single document.

For modeling the structural aspects of a system, UML contains following six diagram types.

- **Class diagram.** A class diagram is used for describing structural concepts existing in a modeled system (i.e., classes), composed of their properties (i.e., attributes), their behavior (i.e., operations), and relationships that may exist between them (i.e., associations).
- **Object diagram.** An object diagram can be used to describe a state of a running system at the certain point in time of the system's execution, consisting of existing

objects in the state, their attribute values, and relationships that exist between them.

- **Package diagram.** A package diagram is used for grouping the components of a modeled system into separate packages.
- **Component diagram.** A component diagram models different components, with defined interfaces and dependencies, out of which a system is composed.
- **Composite structure diagram.** A composite structure diagram can be used for hierarchically decomposing the elements of a modeled system.
- **Deployment diagram.** A deployment diagram can be used for modeling the communication between elements of a modeled system during runtime, as well as specification of the deployment of runtime artifacts, such as files on a server running the system.

For modeling the behavioral aspects of a system, UML contains following seven diagram types.

- **Use case diagram.** A use case diagram can be used to model the functionality provided by a modeled system. It models the interaction of users (i.e., actors) with the system, as well as relationships between users and different use cases of the system.
- **Activity diagram.** An activity diagram is used for describing actions that are executed by a system, as well as the control and data flow between those actions.
- **State machine diagram.** A state machine diagram models the lifecycle of an object consisting of states in which such object can be, the possible transitions between the states, and the actions that can be executed in each state.
- **Sequence diagram.** A sequence diagram can be used to model interactions between objects in a modeled system, needed for accomplishing a certain task.
- **Communication diagram.** Similar to sequence diagram, it is used for modeling interactions between objects in a modeled system, however describing the structural relationships between interaction partners.
- **Timing diagram.** A timing diagram can be used for modeling the state changes of interacting objects in a modeled system during an interaction.
- **Interaction overview diagram.** An interaction overview diagram describes the coordination of different interactions by visualizing in which order interactions can take place.

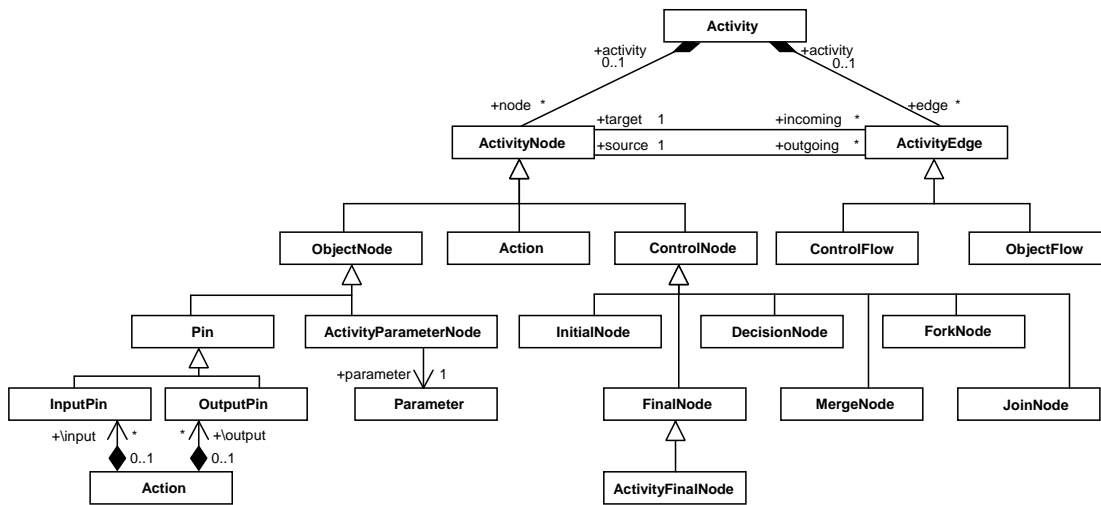


Figure 4.1: Excerpt of the UML metamodel containing the basic concepts of activity diagrams [Obj15]

The abstract and concrete syntax of UML is precisely and completely specified by the UML OMG standard [Obj15]. However, the execution semantics of UML is neither precisely nor completely specified. It is only informally defined in English prose and scattered throughout the standard. This leads to different interpretations of the same specification by different individuals.

In order to address this issue, a standard called *Semantics of a Foundational Subset for Executable UML Models* (fUML) [Obj11] was introduced by OMG.

The fUML standard defines precise execution semantics for a subset of UML containing the most relevant part of class diagrams for modeling the structure, and activity diagrams for modeling the behavior of a software system. This subset was designed with intention to build a foundation with which the execution semantics of higher level UML modeling concepts could be defined.

## 4.2 Modeling Behavior with fUML Activities

Activity diagrams are used for describing behavior of a modeled system. An activity diagram can describe a workflow at a very high level of abstraction, as well as low level operations on objects and their properties.

An excerpt of a metamodel containing the basic concepts for modeling activities is depicted in Figure 4.1.

Activities are composed of *activity nodes* and *activity edges*. *Actions* are activity nodes modeling a single step in the activity, such as processing data or control flow. Actions


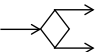
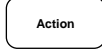

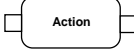

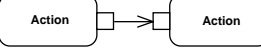

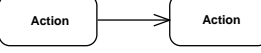


	Activities describe the behavior of a system.		Decision nodes are used to define alternative branches and the guard conditions that specify under which conditions branch has to be chosen.
	Actions represent the individual steps necessary to accomplish an activity.		Merge nodes merge alternative branches.
	Actions can have inputs which are modeled using input pins and outputs which are modeled using output pins.		Fork nodes are used to model concurrent branches.
	An object flow edge describes the data flow between actions.		Join nodes join concurrent branches.
	A control flow edge describes the control flow through the actions.		The initial node is used to specify the starting point of an activity.
	The final node specifies the end of an activity and therewith the end of every control or data flow.		

Figure 4.2: Modeling concepts of UML activity diagrams [May11]

can have inputs and outputs modeled using so called *input pins* and *output pins*. For modeling input or output of an activity, *activity parameter nodes* can be used.

To define start or end of an activity, and alternative or concurrent branches, control nodes can be used. *Initial node* is used for modeling the start of an activity. End of an activity can be modeled using *activity final node*.

For modeling concurrent branches within an activity, *fork* and *join* nodes can be used. Alternative branches are modeled using *decision* and *merge* nodes.

Activity nodes within an activity are connected by *activity edges*. *Control flow* edges are used for modeling the control flow within an activity. Data flow within an activity is modeled using *data flow* edges. In Figure 4.2, diagram notation of modeling concepts of UML activity diagrams are presented.

For understanding the activity diagram execution semantics, it is important to understand the concept of *tokens* originating from Petri Nets [Rei85]. Tokens are used to describe possible execution flows of an activity. There is no additional modeling notation for tokens, and they are not part of the UML metamodel.

Tokens flow along activity edges from one activity node to another, possibly carrying objects representing the data flow within the activity. Tokens carrying data are called *data tokens* or *object tokens*. Tokens which do not carry data are called *control tokens*.

An activity node can be executed if all incoming edges of that node have received a token. Once an activity node is executed, tokens are provided on all outgoing edges of that node. There can be more than one token in an activity during execution, e.g., when fork nodes are executed, or when there are more than one initial node in the activity.

UML provides predefined *primitive actions* for modeling the manipulation of objects and links, and communication among objects [May11]. These actions are primitive enough to be interpreted and executed by a computer, and thus they are included in the fUML



UML	Included	fUML	Included
<i>Executable Nodes</i>			
Structured Activity Node	Yes	Conditional Activity Node	Yes
Loop Node	Yes	Sequence Node	No
Expansion Region	Yes		
<i>Control Nodes</i>			
Initial Node	Yes	Activity Final Node	Yes
Decision Node	Yes	Merge Node	Yes
Fork Node	Yes	Join Node	Yes
Flow Final Node	No		
<i>Object Nodes</i>			
Activity Parameter Node	Yes	Expansion Node	Yes
Central Buffer Node	No	Data Store Node	No
<i>Activity Edges</i>			
Control Flow	Yes	Object Flow	Yes
<i>Object Related Actions</i>			
Create Object Action	Yes	Destroy Object Action	Yes
Read Self Action	Yes	Test Identity Action	Yes
Reclassify Object Action	Yes	Read Is Classified Object Action	Yes
Read Extent Action	Yes	Start Classifier Behavior Action	Yes
Start Object Behavior Action	Yes		
<i>Link Related Actions</i>			
Create Link Action	Yes	Create Link Object Action	No
Read Link Action	Yes	Read Link Object End Action	No
Read Link End Qualifier Action	No	Clear Association Action	Yes
Destroy Link Action	Yes		
<i>Variable and Structural Feature Related Actions</i>			
Add Variable Value Action	No	Read Variable Action	No
Clear Variable Action	No	Remove Variable Value Action	No
Add Struct. Feature Value Action	Yes	Read Struct. Feature Value Action	Yes
Clear Struct. Feature Action	Yes	Remove Struct. Feature Value Action	Yes
Value Specification Action	Yes		
<i>Communication Related Actions</i>			
Accept Call Action	No	Accept Event Action	Yes
Call Behavior Action	Yes	Call Operation Action	Yes
Broadcast Signal Action	No	Send Signal Action	Yes
Send Object Action	No	Reply Action	No
<i>Other Actions</i>			
Opaque Action	No	Raise Exception Action	No
Reduce Action	Yes	Unmarshall Action	No

Table 4.1: Modeling concepts of UML activities and actions language included in fUML subset version 1.0 [May11]

standard. Comparison of available activities and actions language of UML and fUML, showing which primitive actions and control nodes of UML metamodel are included in the fUML standard, is presented in Table 4.1.

The fUML standard, as stated before, defines a subset of the UML modeling concepts and a precise execution semantics for the selected elements. The package structure of fUML metamodel is same as the package structure of UML. Packages that are not included

Modeling of Structure			
Classes	Yes	Components	No
Composite Structures	No	Deployments	No
Modeling of Behavior			
Actions	Yes	State Machines	No
Common Behaviors	Yes	Interactions	No
Activities	Yes	Use Cases	No

Table 4.2: UML packages included in fUML subset version 1.0 [May11]

in fUML are entirely excluded. Those packages that are included may be restricted compared to the corresponding packages in UML, i.e., some elements of a package may be excluded and additional constraints may be defined.

In Table 4.2 the packages of the UML metamodel which are included in fUML are presented. The packages are grouped into two categories: the packages for structural modeling and the packages for behavioral modeling.

The fUML standard comprises a foundational core of UML composed of modeling concepts for describing the structure and the behavior of a system. For describing the structure classes are used, and for describing the behavior of the system activities are used. This subset constitutes the *abstract syntax* of fUML, on which additional wellformedness rules in form of OCL constraints are imposed. These OCL constraints are necessary for precisely defining the semantics of the modeling concepts in the subset.

There are three available alternatives for representing fUML models. One is use of *graphical notation* defined by UML standard, i.e., class and activity diagram notations. Another alternative is representation of fUML models in the *textual concrete syntax* defined by the Alf standard [Obj13a]. Finally, a *mixture* of the previous two can be used.

### 4.3 Execution Semantics of fUML

The semantics of fUML is defined by an *execution model* specifying a virtual machine capable of interpreting the fUML models. A reference implementation of the fUML virtual machine was implemented by Model Driven Solutions<sup>1</sup> using Java programming language, and is available under the Academic Free License version 3.0.

#### 4.3.1 The Execution Model of fUML

The execution model of fUML is realized in an operational way. In particular, an interpreter for fUML models is defined using a subset of fUML called base UML subset (bUML). The execution semantics of bUML in turn is specified with the first order logic formalism Process Specification Language (PSL) [Int04].

Beside the packages containing concepts for modeling structure and behavior of a system, fUML also comprises packages defining the execution model that specifies the execution

<sup>1</sup><http://portal.modeldriven.org/>

semantics of all concepts included in fUML, as well as an execution engine and an execution environment that allow the execution of fUML models. The execution model is structured into the following packages.

- **Loci.** This package specifies the execution engine and the environment
- **Classes.** This package defines the structural semantics of fUML.
- **Common Behaviors, Activities, Actions.** These packages define the behavioral semantics of fUML.

The execution engine offers three operations.

- *Execute.* To execute a behavior synchronously, the execute operation can be used. For this operation input can be provided, and output is returned.
- *Evaluate.* This operation is used for evaluating value specifications, and returning the corresponding values.
- *Start.* To execute a behavior asynchronously, the start operation can be used. The operation returns a reference to the instance of the execution behavior.

To execute an fUML activity model, the execution engine performs the following steps [May11].

1. **Input values are provided to the input activity parameter nodes.** In order to execute the activity, all its input parameter nodes have to obtain values.
2. **The enabled nodes are identified.** In this step, nodes which can be executed are identified. At the beginning of the execution, nodes which are enabled are initial nodes, activity input parameter nodes, and actions without any incoming edges.
3. **Distribution of control tokens to enabled nodes.** Control tokens are distributed to the identified enabled nodes from the previous step.
4. **Execution of enabled activity nodes.**
5. **Output values are provided to the output activity parameter nodes.** Once there are no more enabled nodes in the activity, execution terminates, and values are provided to all output parameter nodes of the activity.

Furthermore, the fourth step of execution of an fUML activity can be divided into following steps.

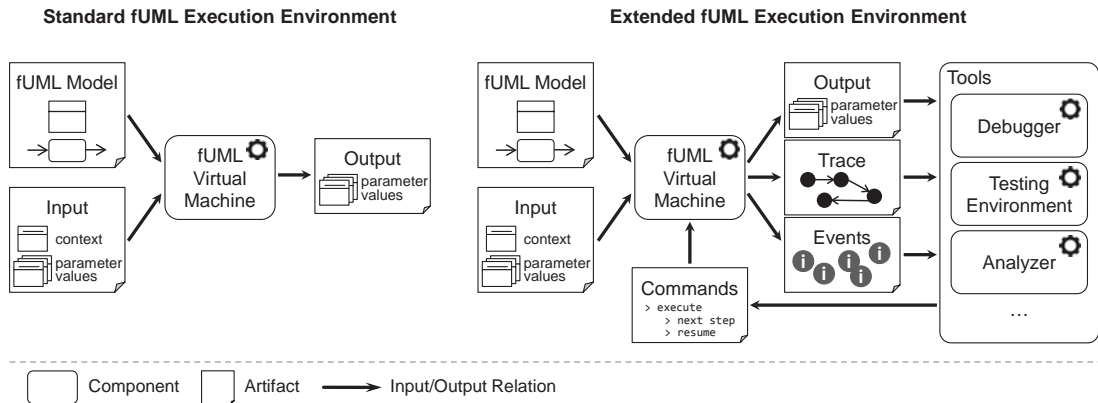


Figure 4.3: Overview of fUML execution environment extensions [May14]

1. **Check if enabled activity node can be executed.** It is checked if control tokens are available on all incoming control flow edges of the activity node, and if the necessary data tokens are provided to all input pins.
2. **Consumption of the tokens.** Once an activity node can be executed, all tokens from incoming edges are removed and added to the activity node.
3. **Execution of the activity node.** Once the tokens are consumed, the behavior of the activity node is executed. Any generated data tokens are provided to the corresponding output pins of the activity node.
4. **Sending of tokens to the subsequent activity nodes.** All outgoing control flow edges of the executed activity node receive a control token. In case any data is generated, data tokens are sent through the output pins to the outgoing object flow edges and, subsequently to successor nodes.
5. **Check if the activity node should execute again.** Once the activity node is executed, and all control and data tokens are sent, it is checked if the executed activity node can be executed again by repeating previous four steps.
6. **Execution of subsequent activity nodes.** Previous five steps are performed for all activity nodes that received tokens in the previous step.

### 4.3.2 Extensions of the fUML Execution Environment

The fUML virtual machine takes as input an fUML model, a reference to the activity that shall be executed, as well as input parameter values and a context object for this activity. After the execution, it provides as output the end result of the execution comprising the output parameter values obtained for the executed activity. The execution process is depicted on left side of Figure 4.3.

As can be seen from Figure 4.3, the standardized fUML execution environment comprising the fUML virtual machine does not provide means for the analysis methods, such as testing, debugging, and non-functional property analysis. In order to support these analysis methods, the virtual machine needs to contain following characteristics [May14].

**Observability.** The ability to monitor the state of an execution being carried out by the virtual machine. This characteristic is important for analysis methods such as debugging, non-functional property analysis, profiling and monitoring.

**Controllability.** The ability to control executions being carried out by a virtual machine. This characteristic is necessary for analysis methods such as debugging and non-functional property analysis.

**Analyzability.** The ability to analyze ongoing or completed executions based on captured runtime information. This characteristic is essential for supporting analysis methods such as testing, non-functional property analysis, and evolution (e.g., version differencing).

To overcome the limitations of the standardized virtual machine, and support the necessary characteristics for the analysis methods, an extension of the fUML execution environment has been developed [MLK12]. An overview of the extensions is depicted on the right side of Figure 4.3. The extensions comprise an *event mechanism*, issuing events for notifying about the state changes of an ongoing model execution, a *command interface* to the fUML execution environment, enabling the issuing of commands for controlling the execution of a model, and a *trace model* capturing the execution traces of fUML activities, which provides ability to dynamically analyze partially or completely performed execution of a model.

For building our testing framework, the *trace model* is essential. Further details on the trace model, and how it is used by the testing framework, are given in Chapter 7.



# Testing Framework

In this chapter, we introduce an example of a UML model specifying an automatic teller machine (ATM) system, which serves as running example throughout this and the two following chapters. Based on the example, we derive the functional requirements for our testing framework. Finally, at the end of the chapter an overview of the framework, describing the process of specifying and executing the test cases, is given.

## 5.1 ATM Example

The structure of an ATM system, where a client can withdraw or deposit an amount of money from (to) his or her account, is depicted in Figure 5.1.

The ATM system provides several functionalities. A user can withdraw or deposit an amount of money from (to) his or her account (*withdraw* and *deposit* operations of the ATM class). For performing the withdraw or deposit process, a transaction has to be created and maintained during the process, and finally closed and recorded once the process is completed (*startTransaction* and *endTransaction* operations of the ATM class).

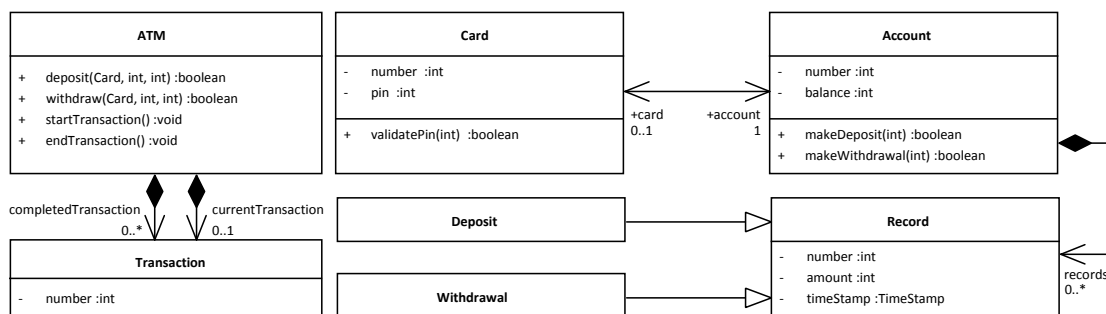
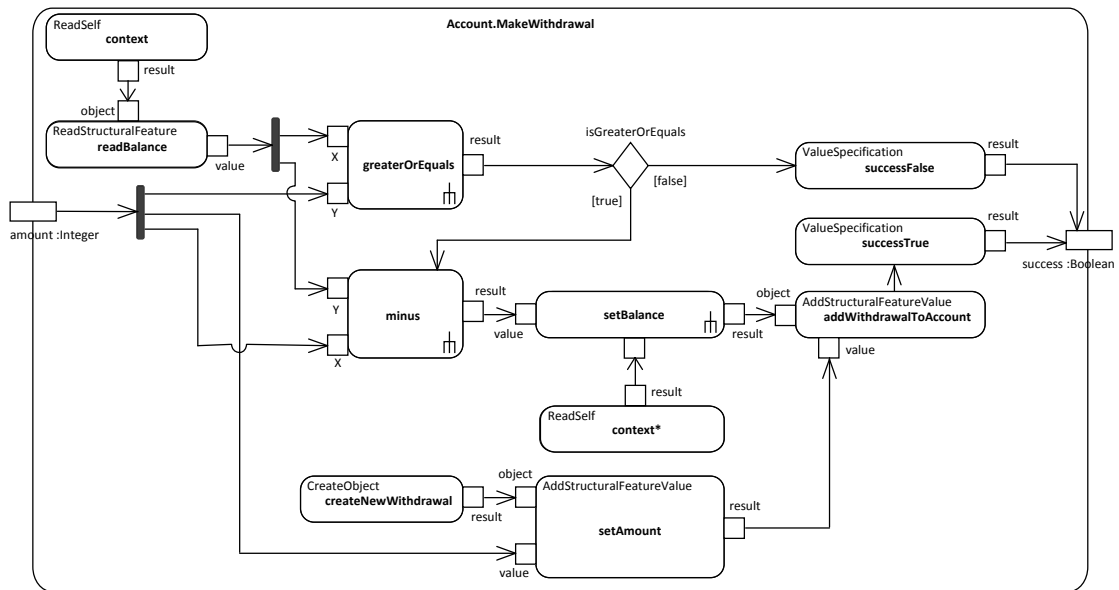


Figure 5.1: Structure of the ATM system

Figure 5.2: Activity *MakeWithdrawal* of the Account class

The ATM keeps track of all processed transactions, as well as the transaction currently being processed (*completedTransactions* and *currentTransaction* associations between ATM and Transaction classes). In order to perform a deposit or withdrawal in the ATM system, user needs to use a card, represented by the class *Card*. The card has a number and a pin, used for validating the identity of the user.

The operation *validatePin* of the *Card* class specifies the validation process. Each card is associated with an account (association *account* between classes *Card* and *Account*), for which a deposit or a withdrawal can be performed. Operations *makeWithdrawal* and *makeDeposit* of the *Account* class specify the process of making a withdrawal or a deposit on the account. For each withdrawal and deposit created for the account, a list of records (association *records* between *Account* and *Record* classes) is kept.

Activity *MakeWithdrawal* implementing the *makeWithdrawal* operation of the *Account* class is depicted in Figure 5.2.

As the first step, the balance of the account is read (*context* and *readBalance* actions) and it is checked whether it is greater or equal to the amount of money to be withdrawn from the account, provided by the user (action *greaterOrEquals*). If the amount of money to be withdrawn is greater than the balance of the account, a false value (*successFalse* action) is provided as the output of the activity.

Otherwise, the difference between the balance of the account and the provided amount is computed (*minus* action), and set as the new balance of the account (*context\** and *setBalance* actions). A new withdrawal record is created (*createNewWithdrawal* action), and its amount is set to the value of the provided amount by the user (*setAmount*



action). Finally, the new withdrawal record is added to the list of records of the account (*addWithdrawalToAccount* action), and a true value is returned as output of the activity (*successTrue* action).

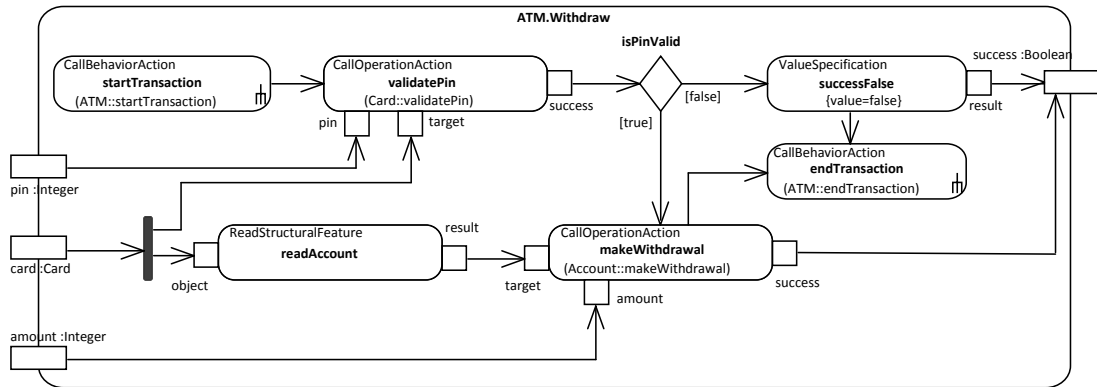
Actions *readBalance* and *setBalance* are *readStructuralFeature* and *addStructuralFeature* actions respectively, invoked on the object provided as output of the *context* action. Here, the action *readBalance* retrieves the balance (an integer value) of the current account object for which the activity *MakeWithdrawal* was invoked, and sends it to its output pin *result*. Furthermore, the action *setBalance* sets the balance of the current account object to the integer value provided to its input pin *value*, and provides the account object to its output pin *result*, with the new balance set. The *context* action is a *readSelf* action, used to retrieve the object which is set as context of the activity. This action is defined only once in the activity, however, for readability it is presented twice in the diagram of the activity.

In order to ensure the functional correctness of performing a withdrawal from an account, the activity *MakeWithdrawal* of the ATM system has to fulfill a number of functional requirements:

1. The amount of money to be withdrawn from the account should be checked before calculating the new balance of the account and before calculating the output of the activity indicating whether the withdrawal was successful.
2. If the amount of money to be withdrawn does not exceed the balance of the account:
  - a) the balance of the account should be updated accordingly,
  - b) a new withdrawal record should be created and added to the records of the account, and
  - c) a true value should be returned as output of the activity.
3. If the amount of money to be withdrawn exceeds the balance of the account:
  - a) the balance of the account should remain the same,
  - b) the number of records of the account should remain the same,
  - c) no new withdrawal record should be created in the system, and
  - d) a false value should be returned as output of the activity.

Activity *ATM.Withdraw* implementing the *withdraw* operation of the ATM class is depicted in Figure 5.3. To perform the withdraw process, as a first step a new transaction is created and set as the current transaction of the ATM system (*startTransaction* action). For the pin provided by the user it is checked whether it corresponds to the pin of the card (*validatePin* action). If the provided pin does not correspond to the pin stored on the card (i.e., it is not valid), a false value is provided as output of the activity (*successFalse* action).

Otherwise, the account is retrieved (*readAccount* action), the withdrawal is performed resulting in an update of the account's balance, and a withdrawal record is created and

Figure 5.3: Activity *Withdraw* of the ATM class

added to the list of records of the account (*makeWithdrawal* action). Finally, the current transaction is ended and added to the completed transactions in the ATM system (action *endTransaction*). The actions *startTransaction* and *endTransaction* are call behavior activities, used to invoke activities specifying processes of setting and removing the current transaction of the ATM system represented by the operations *startTransaction* and *endTransaction* of the ATM class.

To ensure the functional correctness of handling withdrawals in the ATM system, the activity *ATM.Withdraw* has to fulfill following functional requirements:

1. At the beginning of the withdrawal process, a new transaction is created, set as the current transaction of the ATM, and maintained until the end of the withdrawal process.
2. If an invalid pin is provided, or the amount of money to be withdrawn exceeds the balance of the account:
  - a) the number of the withdrawals and the balance of the account should remain the same, and
  - b) a false value should be returned as output of the activity.
3. If a valid pin is provided and the amount of money to be withdrawn from the account does not exceed the balance of the account:
  - a) the withdrawal records and the balance of the account should be updated accordingly, and
  - b) a true value should be returned as output of the activity.
4. For each execution of a withdrawal:
  - a) after the update of the account's balance or the non successful validation of the pin or the balance, the current transaction of the ATM should be removed and added to the completed transactions, and

- b) the balance of the account should be equal to the difference of the sum of all deposits and the sum of all withdrawals.

Based on the presented functional requirements for the ATM system, in the next section we will derive and present requirements for our testing framework.

## 5.2 Testing Framework Requirements

In order to be able to test the functional requirements of the ATM system, our testing framework has to provide several capabilities.

Looking back at the requirement 1 from Section 5.1 for the *Account.MakeWithdrawal* activity, it is stated that the elements of the activity performing the check whether the amount of money to be withdrawn from the account does not exceed the current balance of the account should always be executed before the elements performing the calculation of the new balance after the withdrawal, as well as before calculating the output of the activity. Therefore, we need a way to specify that the action *greaterOrEquals*, comparing the amount provided as input of the activity with the balance of the account, should be executed before the action *minus*, which calculates the difference between the balance of the account and the provided amount. Also, it has to be checked that the actions *successFalse* and *successTrue* calculating the output of the activity are executed after the action *greaterOrEquals*. Moreover, we are not concerned with activity nodes which might be executed before, after or between the actions *greaterOrEquals* and *minus*, therefore we need a way to specify a relative order of node executions concerning specific nodes of an execution path.

Furthermore, the need to specify a relative execution order of nodes also comes from the fact that in an activity it is possible to specify concurrent flow of control and data, e.g., by using fork and join nodes, or by specifying several starting nodes, as is the case in the activity *Account.MakeWithdrawal*. This can lead to the existence of a number of possible execution paths of the activity under test, which all have to be taken into account when evaluating the execution order of nodes. Therefore, in order to define this requirement, our testing framework should provide means for specifying a relative execution order of nodes within an activity under test, where only the order between selected nodes is indicated, while the rest of the nodes are ignored.

In the requirements 2a and 3a of the *Account.MakeWithdrawal* activity, it is stated that depending of whether the withdrawal amount does or does not exceed the balance of the account, the balance should be updated accordingly or remain the same, once the activity is executed. Therefore, it should be possible to specify assertions checking the value of a property of an object manipulated by an activity at a certain point in time of the activity execution.

The requirements 2b and 3b state that, based on whether the amount to be withdrawn exceeds the balance of the account, a new withdrawal record should or should not be

created and associated with the account, once the activity is executed. In order to be able to define these requirements, our testing framework needs to provide means for specifying assertions regarding links between objects, that are processed by an activity under test. For these requirements, we would need to specify that the number of withdrawals associated with the account, is increased by one in case of a successful withdrawal or remains the same in case of an unsuccessful withdrawal.

The requirements 2c and 3c for the *Account.MakeWithdrawal* activity specify that a true or false value should be provided as output of the activity once the execution is completed, depending on whether the amount to be withdrawn exceeds the balance of the account. Therefore, in order to be able to define these requirements, our testing framework should provide means for specifying assertions on the values provided as output of the activity under test.

Requirement 1 for the *ATM.Withdraw* activity states that an instance of the Transaction class should be created and set as the current transaction of the ATM object. Stating that an instance of a class should exist within the system at the certain point in time could be specified as an assertion on a complete state of the system, rather than as an assertion of a specific object processed by the activity under test. Therefore, it is necessary to provide means for specifying assertions on the complete execution state of an activity under test at the certain point in time. This is also necessary for specifying a test case checking the fulfillment of Requirement 3c of the *Account.MakeWithdrawal* activity.

Requirements 2 and 3 for the *ATM.Withdraw* activity are similar to the requirements 2 and 3 for the *Account.MakeWithdrawal* activity, and can be expressed using the same means. However looking back at the requirement 4b for the *ATM.Withdraw* activity, it might be necessary to express complex assertions involving iteration and computation over objects and their properties, existing in an execution state of the system at the certain point in time. In this requirement, it is stated that the balance of the account should be equal to the difference between the sum of all deposits and the sum of all withdrawals associated with the account. Therefore, we need a way to specify an expression iterating over all records of the account, summing up all amounts of deposit and withdrawal records respectively, and calculating their difference.

Having a look at the requirements 1 and 4a of the *ATM.Withdraw* activity, we can observe that a testing framework for fUML models has to provide another capability. These two requirements together state that for each execution of the withdrawal process, a new transaction has to be created that is maintained during the withdrawal process, and ended and recorded at the end of the process. To be able to assert this requirement, we have to be able to check the state of the system at different points in time of the execution as well as the states of the system between certain time frames within the execution.

The selection of the states to be checked may be based on the execution of certain activity nodes or based on conditions that are fulfilled by these states. For instance, for

checking requirement 1, we have to specify the time frame between starting and ending a new transaction. This can be done based on the state of the system. In particular, a transaction is started as soon as the current transaction of the ATM is set, and ended as soon as the current transaction is again unset. Similarly, for checking requirement 4a, we have to identify the last execution state of the system within the withdrawal process. This is the state of the system after the execution of the action *endTransaction*.

The requirements for our testing framework are summarized as follows.

*TF-R1* The testing framework should provide the possibility to test the chronological order in which nodes of the activity under test should be executed. Thereby, the framework ensures that the specified order is correct for each possible execution of the activity, taking into account parallelism.

*TF-R2* The testing framework should enable to check whether for a given activity input, the correct output is produced. Also, it should enable to check the inputs and outputs of actions within the activity under test.

*TF-R3* It should be possible to check the state of the system during the execution of the activity under test composed of objects existing in the respective state:

- a) by selecting the objects to be checked, and
- b) by checking either their property values directly or evaluating complex constraints over the whole state.

*TF-R4* Furthermore, it should be possible to select the states to be checked through the definition of time frames

- a) by referring to actions within the activity under test denoting the start and end of the time frame to be considered, or
- b) by specifying constraints that should be satisfied by the states to be checked.

## 5.3 Testing Framework Overview

The process of testing fUML activities with our testing framework, depicted in Figure 5.4, is performed in three main phases: *test creation phase*, *activity execution phase*, and *test evaluation phase*.

In the test creation phase, a test suite is specified using the test specification language. A test suite is composed of test scenarios and test cases.

A test scenario is composed of a set of objects and links describing a state of the system under test. A scenario may be used for specifying the initial state of the system under test, or can be used for asserting (1) the output of an activity under test or (2) an intermediate states during the execution of the activity. Furthermore, the initial state of the system can be created by composing several test scenarios within a test case.

A test case specifies the activity under test, and data provided as input to the activity under test. Objects and links defined within scenarios can be provided as the input data for an activity under test within the test case. Finally, a test case is composed of a set of assertions for validating the execution states of the activity under test, or for validating the execution order of activity nodes.

The state assertions may specify an expected output of an action within the activity under test, or the activity itself. Furthermore, it is possible to specify OCL [Obj12] expressions that should be evaluated at a certain point in time of the activity execution.

Order assertions can be used to specify absolute or relative orders of node execution within the activity under test. The order assertion is specified by listing the nodes of the activity under test, in the order in which they are expected to be executed, separated by a comma. For specifying relative order in an order assertion, where one or more nodes in the order assertion are skipped, special escape characters are available. More details regarding the test specification language, and how to specify test scenarios and test cases, are given in Chapter 6.

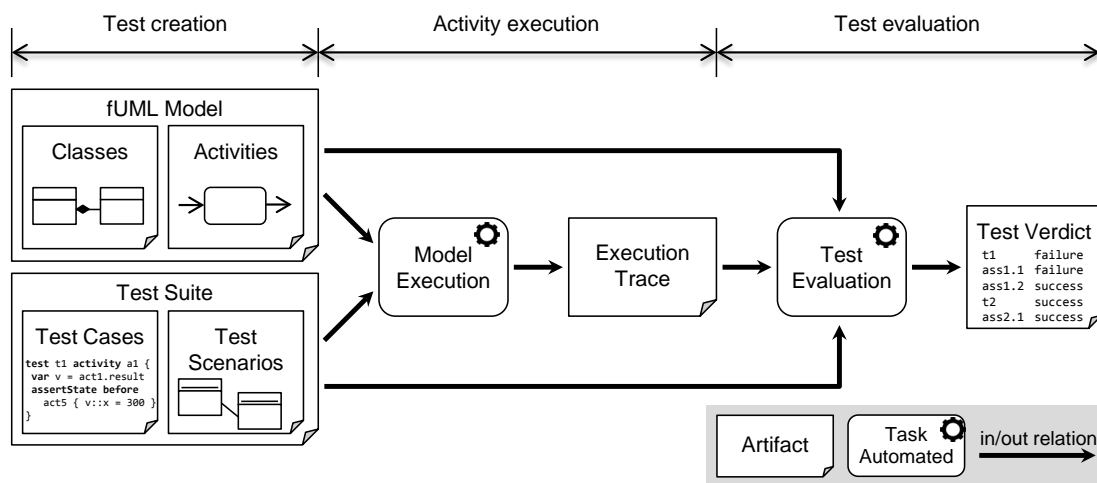


Figure 5.4: Overview of the testing framework

In the activity execution phase, the test input data specified in test scenarios is converted into the fUML representation and, together with the activities under test, provided as input to the fUML virtual machine. The fUML virtual machine executes the activities under test, and thereof produces execution traces, containing information needed for evaluating the test cases.

The execution trace contains information such as for instance the logical and chronological dependencies between executed activity nodes, as well as information regarding which objects and links were created or destroyed by which actions within the activity under test. The fUML virtual machine and the execution trace produced by it are described in Chapter 4 and Chapter 7, respectively.

In the test evaluation phase, the execution trace and the test cases are provided to the test interpreter. The interpreter evaluates each assertion by analyzing the execution trace and produces the test verdict. The test verdict contains information regarding the executed activity under test, the input provided to the activity under test, and a set of results for each specified assertion.

In case of a failing order assertion, the number of execution paths which are not valid according to the specified order, as well as several invalid execution paths are presented. If a state assertion specifying an expression regarding inputs or outputs of a node within the activity under test, or the inputs or outputs of the activity itself fails, the expected and the real values are presented. In case of an OCL constraint violation, the name of the constraint is presented. The details concerning the test evaluation phase are presented in Chapter 7.





# Test Specification Language

In this chapter we give a detailed overview of our test specification language. Each concept of the test specification language, such as test scenarios and test cases, are presented in detail in Section 6.1. In the following Section 6.2 we revisit the ATM example from Chapter 5, and we specify and present test cases derived from the presented functional requirements for the ATM model. Finally, in Section 6.3 we give an overview of the Xtext framework used for implementing our test specification language.

## 6.1 Test Language Concepts

In order to support testing of fUML activities, we have designed and developed a test specification language and an environment containing an editor for specifying test cases on the model level. Our test specification language supports specifying assertions on the execution order of nodes contained by the activity under test, as well as assertions on the state of the system under test.

To test an activity, it might be necessary to provide input values to all its input parameters (cf. Chapter 4). A test designer might specify input for an activity under test by creating a *test scenario*, composed of objects and links that can be used in the test cases. How to specify a test scenario, comprising objects and links, is described in Subsection 6.1.2.

The main component of the test specification language is a *test case*. A test case is used for validating an activity under test for a defined input, and is composed of a number of assertions for validating the execution order of activity nodes, as well as for checking the state of the system during the execution of the activity under test. Details on specifying a test case are given in Subsection 6.1.3.

A test designer might be interested in asserting complete or partial execution orders of nodes of an activity under test (*TF-R1* from Section 5.2). For instance, the functional requirement 1 of the *Account.MakeWithdrawal* activity of the ATM system from

Chapter 5 can be specified as an order assertion where it is checked whether the action *greaterOrEquals* performing the validation of the amount of money to be withdrawn from the account is always executed before the action calculating the new balance of the account.

Furthermore, an activity under test might call other activities, whose order of execution might be of interest in a specific test case. Our test specification language supports specifying relative execution orders of activity nodes, as well as specifying assertions on the execution order of nodes (so-called suborders) contained by activities that are called by the activity under test. Details on specifying and using order assertions in the test specification language are given in Subsection 6.1.4.

The execution of an action within an fUML activity might lead to the creation or modification of objects in the current system state, resulting in the creation of a new state of the system. Each action within the activity under test, which modified the system state in some way, is a creator of a new state within the trace of execution. A test designer might be interested in specifying assertions on the objects and links created as output of the activity under test, or as output of an action within the activity under test (*TF-R2* from Section 5.2). For instance, the functional requirement 2c for the *Account.MakeWithdrawal* activity of the ATM system from Chapter 5 can be specified as an assertion of the value provided as output of the activity under test. Details on how to specify and use *state assertions* for checking the execution states are given in Subsection 6.1.5.

In order to support validation of complex assertions on the system states, involving for example iterations or calculations over instance property values in a certain execution state, we have added support for evaluation of OCL [Obj12] constraints in our test specification language (*TF-R3* and *TF-R4* from Section 5.2). For example, specification of the functional requirement 4b of the *ATM.Withdraw* activity from Chapter 5 requires complex expressions for selecting objects from an execution state, namely all deposit records and withdrawal records associated with a given account and calculating the sum of their property values, i.e. the sum of all deposit and withdrawal amounts. Details on specifying and using OCL constraints in the test specification language are given in Subsection 6.1.5.

### 6.1.1 Import Statement

An *import statement* is used for importing elements of a model under test. An example of an import statement is presented in Listing 6.1.

Listing 6.1: Import statement specification

```
1 import package_name.*
```

In Listing 6.1, all elements of the package with the name *package\_name* are imported into the test suite.

### 6.1.2 Test Scenario

In order to execute the activity under test, it might be necessary to provide values for all activity input parameters as well as a context object. For specifying objects and links of existing classes and associations in a model under test, which can be used as input or context of an activity under test, a test scenario can be defined.

A *test scenario* is composed of a name, definitions of objects, and definitions of links. In Listing 6.2, an example of a test scenario is presented.

Listing 6.2: Test scenario specification

```

1 scenario scenario_name[
2   object object_name: class_name {property_name = 'string'; ...}
3   link association_name {
4     source association_end = object_name;
5     target association_end = object_name;
6   }
7 ]

```

The definition of an object is composed of a name (*object\_name*), type of the object (*class\_name*), and property value declarations. Property value declaration is composed of the name of the property (*property\_name*), assignment operator (=), and a value declaration (e.g., *'string'*). Property declarations are separated by a colon (;). In the testing language, there are three primitive types of property that can be used:

- **String** - arbitrary sequence of characters declared between single quotation marks (e.g. *'this is a string'*)
- **Boolean** - values *true* and *false*
- **Integer** - signed numerical values (*...*, *-10*, *0*, *10*, *...*)

The definition of a link is composed of type of the link (*association\_name*), source and target end declaration. Declaration of a link end is composed of name of the association end of the link (*association\_end*), assignment operator (=), and the name of the object set as the link end value (*object\_name*).

### 6.1.3 Test Case

A *test case* is the main component of a test suite. It groups a set of assertions on the state of execution or order of execution of a specified activity under test. Furthermore, it specifies which input is provided to the activity under test, i.e., which objects or primitive values are provided to which activity input parameters. An example of a test case declaration is presented in Listing 6.3.

Listing 6.3: Test case specification

```

1 test test_name activity activity_name ([parameter_node_name = value, ...]) [on context_object] {
2   [initialize scenario_name;]
3   // body composed of a set of assertions
4 }

```

As presented in Listing 6.3, a test case is composed of a test name (*test\_name*), the name of the activity under test (*activity\_name*), an optional list of input parameter assignments (*parameter\_node\_name = 'value'*), an optional declaration of a context object (*context\_object*), and a set of assertions.

The keyword *initialize*, as presented in Listing 6.3 in line 2, can be used for loading a test scenario as the initial state of the system. By using the initialize statement, all objects and links defined within a scenario will be loaded as the initial state of the system under test. It is possible to load several scenarios in a single test case, by separating the names of the loaded scenarios with comma in the initialize statement.

Values provided as input to a parameter node of the activity under test can be either a primitive value (String, Integer, or Boolean), or an object defined within a test scenario. If no initialize statement has been defined, only objects provided as input or set as context of the activity under test will exist in the initial state.

In the body of the test case several kinds of assertions can be defined. There are two main types of assertions: *order assertion* and *state assertion*, which are explained in the following sections.

#### 6.1.4 Order Assertion

An *order assertion* is used for asserting the expected order in which the nodes of the activity under test should be executed. It is composed of the keyword *assertOrder* and a list of activity nodes in order in which they are expected to be executed, separated by comma.

To specify a relative order of activity node executions, special escape characters can be used. For skipping exactly one node in an order assertion an underscore character ('\_') can be used. For skipping zero or more nodes in an order assertion, a star character ('\*') can be used. An example of an order assertion is presented in Listing 6.4.

Listing 6.4: Order assertion specification

```
1 assertOrder node_one, node_two, *, node_three, _;
```

In Listing 6.4, it is asserted that the first executed node in the activity under test is *node\_one*, immediately followed by *node\_two*, after which there might be an arbitrary number of nodes executed until *node\_three* is reached. Finally, there is exactly one node executed after *node\_three*, which is also the last executed node of the activity under test.

It is also possible to specify a suborder of nodes of an activity invoked by a call behavior or a call operation action from within the activity under test. An example of specifying a suborder in an order assertion is presented in Listing 6.5.

Listing 6.5: Order assertion specification with a defined suborder

```
1 assertOrder node_one (sub_node_one, sub_node_two, *), node_two, *;
```

In Listing 6.5, a suborder is defined for a node *node\_one* of the activity under test. In this case, the node *node\_one* might be either a call operation or a call behavior action, used for invocation of another activity from within the activity under test. Inside the brackets, next to the call action as presented in Listing 6.5, nodes of the called activity are specified in the order in which they should be executed. Same set of escape characters and rules apply for the suborder specification, as for the order assertion itself.

### 6.1.5 State Assertion

*State assertions* are used for checking the state of the system during the execution of an activity under test. A state assertion is composed of a temporal expression, selecting the subset of the execution states that are to be checked, and the body of the state assertion composed of a set of state expressions for checking the selected states.

Temporal expressions are used for defining a time frame that includes all system states that should be checked by the state assertion. The *time frame* of a state assertion can be defined by referring to the actions within the activity under test, or by referring to constraints that should be satisfied in the states defining the time frame. Examples of the three types of state assertions are presented in Listing 6.6.

Listing 6.6: State assertion time frame specification

```

1 assertState always after|until action action_one [until action action_two] { ... }
2 assertState always after|until constraint 'constraint_one'
3   [until constraint 'constraint_two'] { ... }
4 finally { ... }
```

**Temporal expressions.** In the following, we explain temporal expressions based on the examples presented in Figure 6.1. An execution trace produced by the fUML virtual machine for an activity execution is presented. The execution trace is composed of a set of  $n$  system states ( $S_1$  to  $S_n$ ), each created by the execution of an action within the activity under test (*actionA* to *actionX*). The states are organized in chronological order. More precisely, the state  $S_1$  created by action *actionA* precedes the state  $S_2$  created by action *actionB*. This relation between states is indicated by the arrow labeled with word *time*.

There could be a number of state expressions defined for an execution of the activity under test, and each of these expressions have their value for each state in the execution trace. In Figure 6.1, three hypothetical expressions, denoted with *a*, *b*, and *c* are depicted, together with their values presented in each state. For instance, the value of expression *a* in states  $S_2$  to  $S_n$  is evaluated to true.

*Temporal operators* in combination with the specification of actions (or alternatively OCL constraints) are used to specify a time frame selecting the system states to be checked by the state assertion. We support the temporal operators *after* and *until* defining that all states after or until an action has been executed (or alternatively an OCL constraint evaluates to true) shall be checked (cf. Listing 6.6, lines 1-3). If OCL constraints are used, they are evaluated in each state starting from the first state. Those states in which

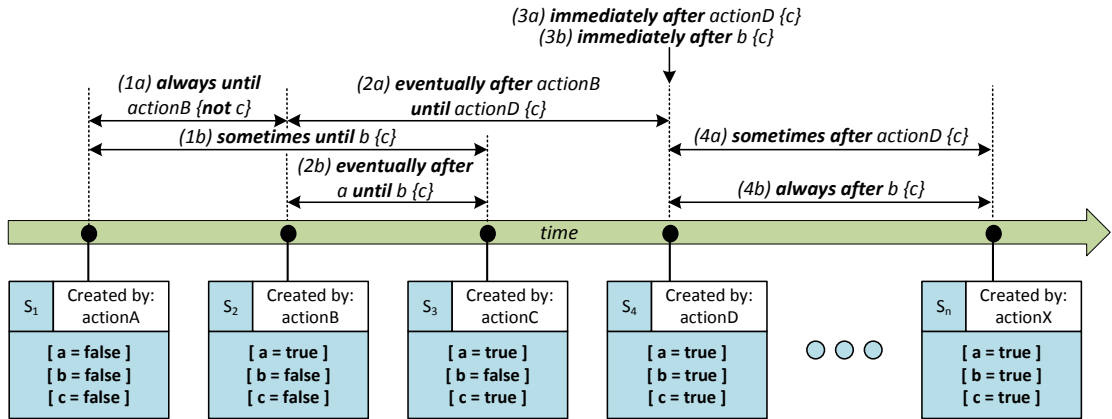


Figure 6.1: Usage of temporal operators and quantifiers in state assertions

the constraints are evaluated to true for the first time, are taken as start and end points of the time frame.

*Temporal quantifiers* are used for specifying in which states of the defined time frame the state expressions of the state assertion should evaluate to true. Our test specification language provides the temporal quantifiers *always*, *eventually*, *immediately*, and *sometimes*, described in the following, based on the examples depicted in Figure 6.1.

If the temporal quantifier *always* is used, each expression in the state assertion should evaluate to true in each state in the time frame. The temporal expression (1a) (cf. Figure 6.1) specifies that in each state starting from the first one until the state produced by `actionB`, the value of the state expression `c` should evaluate to false.

If the temporal quantifier *eventually* is used, it means that each expression should evaluate to true in one of the states in the time frame and should remain true in all following states in the time frame. The temporal expression (2b) (cf. Figure 6.1) specifies that from the first state in which the value of the state expression `a` becomes true, until the first state in which the value of the state expression `b` becomes true, the value of the state expression `c` should become true in one state and remain true in each following state in the time frame.

The temporal quantifier *immediately* is used to specify that each expression should be true in either a state created by the specified action or the one right before it, or if an OCL constraint is used instead, in the first state where the specified constraint is evaluated to true or the state right before it, depending on the temporal operator used. The temporal expression (3a) (cf. Figure 6.1) specifies that the value of the state expression `c` should evaluate to true in the state created by `actionD`.

If the temporal quantifier *sometimes* is used, each expression should evaluate to true in at least one of the states in the defined time frame. The temporal expression (1b) (cf. Figure 6.1) specifies that the expression `c` should evaluate to true in at least one of the

states from the first state until the state where `b` becomes true.

In Listing 6.6 in line 4, *finally* temporal expression is presented. This temporal expression can be used as a shorthand for a temporal expression stating that the last state of the activity execution is to be checked by a state expression.

**State expressions.** For checking the state of an object provided as input or output of an action within the activity under test, or as input or output parameter of the activity itself, an *object state expression* can be defined within a test case. In Listing 6.7, two object state expressions are presented.

Listing 6.7: Object state expression specification

```
1 action_one.result = null;
2 activity_name.result = scenario_name.object_name;
```

An object state expression, as presented in Listing 6.7 in line 1, is composed of a name of an action (*action\_one*), name of the action pin (*result*), an operator, and an object which is compared to the object provided as input or output of the pin.

In Listing 6.7 in line 2 a value provided as input or output from an activity parameter node is checked. To specify the expected object provided by / to a pin or activity parameter node, the objects from a specified test scenario can be used. In line 2 of the listing 6.7, an object with a name *object\_name* defined within the scenario named *scenario\_name* is specified as the expected value of the *result* output pin of the activity under test.

For checking a value of a property of an object provided as input or output of an action of the activity under test, or as input or output parameter of the activity itself, a *property state expression* can be defined within a test case. In Listing 6.8 an example of a property state expression is presented.

Listing 6.8: Property state expression specification

```
1 action_one.result::property_name = 'aString';
```

A property state expression, as presented in Listing 6.8, is composed of the name of an action (*action\_one*) or alternatively the name of the activity, a name of the action pin (*result*) or alternatively the name of an activity parameter node, a double colon separator (`::`), the name of the property to be checked (*property\_name*), an operator, and a value which is compared to the value provided as input or output of the action pin or activity parameter node.

Beside the equality operator, also the following operators are available:

- inequality operator (`!=`),
- comparison operators (`>`, `<`, `>=`, and `<=`) for integer values,
- inclusion operator (*includes*) for checking if a list of values provided as input or output of an action (or activity) contains the declared value, and

- exclusion operator (*excludes*) for checking if a list of values provided as input or output of an action (or activity) does not contain the declared value.

**OCL constraints.** OCL [Obj12] is a formal language providing concepts for defining expressions on UML models. Use cases of OCL include the definition of constraints on UML models, the definition of operation pre and post conditions, as well as the definition of expressions calculating values for derived properties. OCL is like UML standardized by OMG and one of the standards that are part of the OMG MDA framework.

Since OCL is a standardized expression language, targeted at UML, and well-known in the MDE community, we decided to integrate it with our testing framework to enhance the expressiveness of state assertions. In particular, we support the use of OCL for expressing complex conditions on system states in state assertions, as well as for specifying temporal expressions selecting the states to be asserted.

Integrating an OCL constraint in a test case is composed of two parts:

- an invocation of the OCL constraint in a test case, and
- a specification of the OCL constraint itself.

An example of invocation of an OCL constraint is given in Listing 6.9.

Listing 6.9: Invocation of an OCL constraint in a test case

```
1 check 'constraint_name' [on action_one.result];
```

As can be seen from Listing 6.9, an OCL constraint is invoked by its name (*'constraint\_name'*) and an optional context declaration which limits the evaluation of the constraint on a single object (*'on action\_one.result'*). If *'on'* part is left out, the constraint is evaluated on each object in the selected set of states that is of the type declared as the context of the OCL constraint itself.

An example of a specification of an OCL constraint is given in Listing 6.10.

Listing 6.10: Specification of an OCL constraint

```
1 package package_name
2   context class_name
3   inv constraint_name: property_name = 'aString'
4 endpackage
```

An OCL constraint is defined within a package declaration (*package\_name*). The package that has to be declared is the package that contains the context type of the OCL constraint. Each constraint is defined for a context type (*class\_name*), has a name (*constraint\_name*), and a body (*property\_name = 'aString'*). More details about the implementation and usage of OCL constraints in our testing framework are given in Chapter 7.



## 6.2 ATM Example Revisited

In this section we will present one possible implementation of test cases for asserting the fulfillment of the functional requirements of the ATM example presented in Section 5.1 using our test specification language. For each activity presented in Section 5.1, we present a set of test cases implementing the specified ATM functional requirements, and explain in more detail each defined component.

### 6.2.1 Testing the Account.MakeWithdrawal Activity

The activity *MakeWithdrawal* of the *Account* class presented in Chapter 5 implements the process of making a withdrawal for an account in the ATM system. In order to be able to make a withdrawal, an account associated with a card inserted into the ATM has to exist prior to starting the process.

Therefore, in order to test the *MakeWithdrawal* activity of the *Account* class (cf. Figure 5.2), an initial state must be specified. The initial state should contain an instance of the *Account* class, with possibly a number of existing records associated with it. In Listing 6.11 a scenario specifying one possible initial state of the ATM system is presented.

Listing 6.11: Test scenario for testing the *Account.MakeWithdrawal* activity

```

1 scenario BankingTD[
2   object accountTD: Account {balance = 100;}
3   object depositTD: Deposit {amount = 100;}
4   link account_record {source account = accountTD; target records = depositTD;}
5 ]

```

The presented scenario *BankingTD* in Listing 6.11 contains an instance of the *Account* class named *accountTD*, with the property *balance* set to value 100. Furthermore, it contains an instance of the *Deposit* class named *depositTD*, with the same value for the *amount* property. This instance represents a deposit of the account, made by the user at some earlier point in time. Thus, also a link between the objects *accountTD* and *depositTD* is defined in the test scenario.

In order to test the first two functional requirements for the *MakeWithdrawal* activity, we need to write a test case specifying a value to be provided to the input parameter *amount* of the activity, which does not exceed the value of the *balance* property of the *Account* instance. Also, the instance of the *Account* class, for which a withdrawal is to be performed, has to be set as context of the activity. An example test case implementing the first two functional requirements (requirements 1 and 2a-c) for the *MakeWithdrawal* activity is presented in Listing 6.12.

Listing 6.12: *Account.MakeWithdrawal* activity test case: an amount to be withdrawn not exceeding the balance of the account

```
1 test makeWithdrawalSuccess activity Account.MakeWithdrawal(amount=25) on BankingTD.accountTD {
2   initialize BankingTD;
3   assertOrder *, greaterOrEquals, *, setBalance, *, successTrue;
4   finally {
5     context.result::balance = 75;
6     check 'NumOfWithdrawalsSuccess' on context.result;
7     success = true;
8   }
9 }
```

As can be seen in line 1 of Listing 6.12, a value 25 is provided as input to the activity parameter *amount*, specifying the amount of money to be withdrawn from the account. The *readBalance* and *setBalance* actions are read feature value and add feature value actions respectively, which retrieve and set the balance property value of the account object provided to their input pins, respectively. These two actions require that an object of the *Account* class is set as context of the activity under test. Therefore, the context of the activity has been set to the instance *accountTD* of the *Account* class (line 1 in Listing 6.12), previously specified in the test scenario *BankingTD* from Listing 6.11.

In line 2 of the Listing 6.12, an *initialize* statement has been specified. As described in the Section 6.1.3, the initialize keyword enables a test designer to load one or more test scenarios as the initial state of the system under test. In this way, in Listing 6.12 in line 2, we specify an initial state of the system composed of an instance of the *Account* class *accountTD* and an instance of the *Deposit* class *depositTD* with their property values set as specified in the test scenario *BankingTD*, comprising a valid initial state of the *ATM* system.

The functional requirement 1 of the *MakeWithdrawal* activity (cf. Chapter 5) specifies that the validation of the account's balance against the amount of money to be withdrawn has to be performed before the calculation of the new balance of the account. This requirement can be implemented as an order assertion presented in line 3 in Listing 6.12.

Here, we specify that the action *greaterOrEquals*, which performs validation of the balance against the amount to be withdrawn, should be executed before the action *setBalance* updating the balance of the account. As there may be additional nodes executed before, after and in between the relevant actions, we have specified appropriate activity nodes and escape characters in the order assertion.

The functional requirement 2a of the *MakeWithdrawal* activity specifies that the balance of the account should be correctly updated after a successful withdrawal of the account has been made. As the action *context* retrieves the account instance set as context of the activity, the aforementioned requirement can be implemented as a property state expression, asserting the balance property value of the account object provided as output of the context action, at a certain point in time. As the amount of 25 was provided as input of the activity in line 1 in Listing 6.12, and the initial balance of the account is 100 as specified in the test scenario, we assert that the updated balance is 75 in line 4 of

Listing 6.12. As we are interested in the last state of execution of the *MakeWithdrawal* activity, the assertion is specified within the *finally* time expression.

The functional requirement 2b of the *MakeWithdrawal* activity specifies that a new withdrawal record should be created for the account, if the amount of money to be withdrawn does not exceed the balance of the account. To implement this functional requirement we need the ability to specify an expression for evaluating the number of records of the account existing at the end of the activity execution. As previously described in Section 6.1, we have integrated into our test specification language the ability to use OCL [Obj12] for specifying and evaluating complex expressions regarding a set of execution states of the activity under test.

In line 6 in Listing 6.12 an OCL expression named *NumOfWithdrawalsSuccess* is evaluated on the instance of the *Account* class, provided by the output pin *result* of the *context* action. This action is a call behavior action, which receives an integer value through the input pin *value* to be set as new balance of the account instance, and provides the same instance of the *Account* class as its output on the *result* output pin.

The OCL constraint itself is specified in Listing 6.13 in line 6. The constraint selects all instances of the class *Withdrawal* associated with the instance of the *Account* class through the link *records* (i.e., *select(oclIsTypeOf(Withdrawal))*), and invokes the OCL operation *size()* retrieving the number of instances, i.e., withdrawal records found. As there were no previous records of class *Withdrawal* in the initial state of the system (cf. Listing 6.11), the retrieved value of the *size()* operation is asserted to be 1.

Listing 6.13: OCL constraints for testing the *Account.MakeWithdrawal* activity

```

1 context Withdrawal
2 inv NoWithdrawalsCreated: Withdrawal.allInstances() -> size() = 0
3
4 context Account
5 inv NumOfWithdrawalsFail: records -> select(oclIsTypeOf(Withdrawal)) -> size() = 0
6 inv NumOfWithdrawalsSuccess: records -> select(oclIsTypeOf(Withdrawal)) -> size() = 1

```

The functional requirement 2c of the *MakeWithdrawal* activity specifies that a true value should be provided as its output. The implementation of this requirement is presented in Listing 6.12 in line 7. In this state assertion it is checked that a true value is provided as output of the activity via its output parameter *success*.

In Listing 6.14 a test case implementing the functional requirements 3a-d for the *MakeWithdrawal* activity is presented.

Listing 6.14: *Account.MakeWithdrawal* activity test case: an amount to be withdrawn exceeding the balance of the account

```

1 test makeWithdrawalFail activity Account.MakeWithdrawal(amount=150) on BankingTD.accountTD {
2   initialize BankingTD;
3   assertOrder *, greaterOrEquals, *, successFalse;
4   finally {
5     context.result::balance = 100;
6     check 'NumOfWithdrawalsFail' on context.result;
7     check 'NoWithdrawalsCreated';
8     success = false;
9   }
10 }

```

The test case *makeWithdrawalFail* specifies as input an amount of money to be withdrawn from the account (line 1 in Listing 6.14) which exceeds the balance of the account. As explained in the description of the test case in Listing 6.12, the *MakeWithdrawal* activity contains actions which require a context object to be set (i.e., *readBalance* and *setBalance* actions), so a context object for the activity has been set (line 1 in Listing 6.14). Furthermore, using the *initialize* statement (line 2 in Listing 6.14), the scenario from Listing 6.11 is set as the initial state of the system under test.

The functional requirement 3a of the *MakeWithdrawal* activity specifies that the balance of the account should remain the same in case that the amount to be withdrawn exceeds it. Therefore, we have implemented this requirement as a property state expression presented in Listing 6.14 in line 5. The action *context* retrieves the account object set as context of the activity under test. As the value of balance property of the account object was set to 100 in the initial state, specified in the Listing 6.11 in line 2, the assertion specifies 100 as the expected value of the balance property.

The functional requirement 3b of the *MakeWithdrawal* activity specifies that the number of the records of the account should remain the same in case a value exceeding the account's balance is provided as input. The state assertion implementing this requirement is presented in Listing 6.14 in line 6. It calls an OCL expression selecting and counting the instances of the *Withdrawal* class associated with the account. This amount of withdrawal records is asserted to be 0 (constraint *NumOfWithdrawalsFail* from Listing 6.13 in line 5), as there should be no new records associated with the account.

The functional requirement 3c of the *MakeWithdrawal* activity specifies that no new withdrawal record should be created in the system. As the initial state of the system specified with the test scenario in Listing 6.11 contains a single instance of the *Account* class, and no instances of the *Withdrawal* class, in order to specify this requirement, we have to check that once the activity has been executed, in the last system state there should be no existing instances of the *Withdrawal* class, whether they are associated with the instance of the *Account* class or not. Therefore, this requirement can be implemented as an OCL constraint specified in Listing 6.13 in line 2.

The constraint retrieves all instances of the *Withdrawal* class using the *allInstances()* OCL operation, on which the *size()* operation is called to retrieve the number of existing

Withdrawal instances. Finally, this amount is asserted to 0, as there were no instances in the initial state. The state assertion implementing the requirement is presented in Listing 6.14 in line 7.

The last functional requirement 3d of the *MakeWithdrawal* activity specifies that a false value is provided as output of the activity if the amount of money to be withdrawn exceeds the balance of the account. This functional requirement is implemented by the state assertion in Listing 6.14 in line 8, asserting that the activity *MakeWithdrawal* provides a false value for its output parameter *success*.

### 6.2.2 Testing the ATM.Withdraw Activity

The *Withdraw* activity of the ATM system presented in Chapter 5 (cf. Figure 5.3) implements the process of withdrawing an amount of money from the ATM system. The user puts the card into the ATM, provides the pin, and an amount of money to be withdrawn. The system validates the pin provided, and if the validation is successful, invokes the process for making a withdrawal record on the account and updating the balance accordingly (the *MakeWithdrawal* activity in Figure 5.2).

Therefore, in order to test the *Withdraw* activity of the ATM, an initial state containing an instance of the ATM class as well as an instance of the class *Card* must be defined. In addition, the card has to be associated to the account *accountTD* specified in the test scenario *BankingTD* (cf. Listing 6.11) In Listing 6.15 the scenario specifying an initial state of the ATM system required for testing the activity is presented.

Listing 6.15: Test scenario for testing the *ATM.Withdraw* activity

```

1 scenario BankingTDWithAtm[
2   object atmTD: ATM { }
3   object cardTD: Card {pin = 1985;}
4   link card_account {source card = cardTD; target account = BankingTD.accountTD;}
5 ]

```

The scenario is composed of two objects and a link: an object of the class *ATM*, an object of the class *Card* with a value of 1985 set for the property *pin*, and a link between objects *cardTD* and *accountTD*. As can be seen from Listing 6.15, it is possible to associate objects from different scenarios, enabling the composition of input data for test cases. In Listing 6.15 in line 4 a link between the *cardTD* object defined within the scenario *BankingTDWithAtm* and the *accountTD* object defined within the scenario *BankingTD* (cf. Listing 6.11 in line 2) has been specified.

In Listing 6.16 a test case implementing the functional requirements 1 and 2a-b (cf. Chapter 5) for the *Withdraw* activity is presented. The test case specifies an incorrect pin and an amount of money to be withdrawn which exceeds the balance of the account as input for the activity under test. In line 3 in Listing 6.16 an initialize statement is defined, which loads both test scenarios from Listings 6.11 and 6.15, comprising the initial state of the system.

Listing 6.16: *ATM.Withdraw* Activity Test Case: an amount to be withdrawn exceeding the balance of the account, with wrong pin provided

```

1 test atmWithdrawFail activity AIM.Withdraw(card = BankingTDWithAtm.cardTD,
2   pin = 1986, amount = 150) on BankingTDWithAtm.atmTD {
3   initialize BankingTD, BankingTDWithAtm;
4   always after startTransaction until endTransaction {
5     check 'TransactionInProgress';
6   }
7   finally {
8     readAccount.result::balance = 100;
9     check 'NumOfWithdrawalsFail' on readAccount.result;
10    success = false;
11    check 'TransactionEnded', 'TransactionRecorded', 'BalanceRecords';
12  }
13 }

```

The functional requirement 1 of the *Withdraw* activity specifies that a new transaction should be created at the beginning of the withdrawal process, and maintained during its execution. As the actions responsible for creation and removal of the current transaction in the ATM system are actions *startTransaction* and *endTransaction*, the time frame for this assertion is defined as a set of those states which are created between these two actions, and can be defined using the appropriate temporal operators *after* and *until*, as presented in Listings 6.16 and 6.17 in lines 4-6. The OCL expression asserting that the current transaction property of the ATM object is not null (*TransactionInProgress* constraint) is defined in line 2 in Listing 6.18.

The functional requirement 2a of the *Withdraw* activity specifies that in case an invalid pin is provided, or the amount of money to be withdrawn exceeds the balance of the account, the number of withdrawal records and the balance of the account should remain the same. The requirement can be implemented as the state assertion presented in Listing 6.16 in lines 8-9. Here, we specified that the balance of the account provided by action *readAccount* has to be 100, which was the original value specified in the scenario 6.11.

Furthermore, an OCL constraint with name *NumOfWithdrawalsFail* specified in Listing 6.13 in line 5, is checked on the account. As the number of withdrawal records associated with the account were 0 in the initial state, according to the requirement, we assert that this value remained 0.

The functional requirement 2b of the *Withdraw* activity specifies that if an invalid pin, or the amount to be withdrawn exceeding the balance of the account is provided, a false value should be returned from the activity. This functional requirement can be implemented as the state assertion in line 10 in Listing 6.16. Here, we have specified that a false value should be returned from the output pin *success* of the activity.

The functional requirements 1 and 3a-b (cf. Chapter 5) for the *Withdraw* activity are implemented as the test case presented in Listing 6.17.

Listing 6.17: *ATM.Withdraw* Activity Test Case: an amount to be withdrawn not exceeding the balance of the account, with a correct pin inserted

```

1 test atmWithdrawalSuccess activity AIM.Withdraw(card = BankingTDWithAtm.cardTD,
2   pin = 1985, amount = 25) on BankingTDWithAtm.atmTD {
3   initialize BankingTD, BankingTDWithAtm;
4   always after startTransaction until endTransaction {
5     check 'TransactionInProgress';
6   }
7   finally {
8     readAccount.result::balance = 75;
9     check 'NumOfWithdrawalsSuccess' on readAccount.result;
10    success = true;
11    check 'TransactionEnded', 'TransactionRecorded', 'BalanceRecords';
12  }
13 }

```

In the test case, we have specified a correct *pin* of the card, and an amount of money to be withdrawn which does not exceed the balance of the account. In line 3 of Listing 6.17 an initialize statement has been specified, which loads both test scenarios from Listing 6.11 and Listing 6.15, as required objects for testing the *Withdraw* activity are defined within these two scenarios.

As in the previous test case from Listing 6.16, we have specified a state assertion for checking the requirement 1 of *Withdraw* activity in the same way in lines 4-6 in Listing 6.17. The functional requirement 3a specifies that, if a correct pin of the card is provided and an amount of money not exceeding the balance of the account is provided, the number of withdrawal records and the balance of the account should be updated. As the balance of the account was set to 100 in the initial state (cf. Listing 6.11), and the amount to be withdrawn provided in the test case was set to 25, we have implemented this requirement as a state assertion in line 8 in Listing 6.17, asserting the value of the property *balance* of the account instance provided from the output pin *result* of the *readAccount* action to 75.

In line 9 of Listing 6.17 it is asserted that a new withdrawal record has been created. Here, we have specified that the OCL constraint *NumOfWithdrawalsSuccess* should be evaluated on the account instance provided as output of the action *readAccount*. The OCL constraint itself is specified in Listing 6.13 in line 6. As the number of withdrawal records associated with the account was 0 in the initial state, we have specified that it should be updated to 1, as specified in the OCL constraint.

The functional requirement 3b specifies that in case the correct pin and an amount of money to be withdrawn not exceeding the balance of the account are provided, a true value should be returned as output of the activity. This requirement is implemented as the assertion in line 10 in Listing 6.17, asserting that the value provided from output pin *success* of the activity is equal to true.

The last functional requirement 4a-b for the *Withdraw* activity is implemented as three OCL constraints in Listing 6.18 in lines 3-8, and invoked in line 11 in Listings 6.16 and 6.17, respectively.



Listing 6.18: OCL constraints for testing the *ATM.Withdraw* activity

```
1 context AIM
2 inv TransactionInProgress: currentTransaction != null
3 inv TransactionEnded: currentTransaction = null
4 inv TransactionRecorded: completedTransactions -> size() = 1
5 context Account
6 inv BalanceRecords: (records -> select(oclIsTypeOf(Deposit)) -> collect(Record::amount) -> sum()
7   - records -> select(oclIsTypeOf(Withdrawal)) -> collect(amount) -> sum())
8   = balance
```

The functional requirement 4a specifies that once the withdrawal process is completed, the transaction set as current transaction of the ATM is removed from being the current one, and added to the list of completed transactions. This requirement is implemented as OCL constraints *TransactionEnded* and *TransactionRecorded* in lines 3 and 4 in Listing 6.18, respectively.

The functional requirement 4b specified that once the withdrawal process is completed, the balance of the account should be equal to the difference of the sum of all deposits and the sum of all withdrawals. This requirement is implemented as OCL constraint *BalanceRecords* in lines 6-8 in Listing 6.18. Here, we select all instances of the class *Deposit*, and collect all values of the property *amount*, and call the *sum()* OCL operation, to calculate the sum of amounts of all deposit records associated with the account. The same expression is specified and evaluated for the withdrawal records. Finally, the sum of withdrawals is subtracted from the sum of deposits, and asserted to be equal to the value of the *balance* property of the account.

### 6.3 Test Language Implementation

We have implemented an environment for creating test cases with our test specification language using the Xtext framework <sup>1</sup> based on EMF [SBPM08]. Xtext is a framework for building textual software languages, and covers all aspects of a complete language infrastructure, consisting of parser, code generator or interpreter, up to complete Eclipse IDE integration, providing features such as, syntax highlighting, background parsing, error marking, content assist, and quick fixes [Bet13].

Xtext brings many important features to the language infrastructure at runtime, such as *validation*, *linking* and *scoping*. With validation it is possible to check constraints of a developed language, which are not detectable during the parsing phase. Additional checks can be done in a declarative way, providing the errors and warnings which are presented in the IDE. These checks are done during runtime, as the user is creating a model of the developed language, thus immediate feedback is provided. It is also possible to provide quick fixes corresponding to generated errors and warnings during validation.

Xtext contains a generator that takes as input a grammar of a specified language, and will generate artifacts related to the UI editor for the specified language, and more

---

<sup>1</sup><http://www.eclipse.org/Xtext/>



importantly will generate an ANTLR specification from the grammar with all the actions needed to create the abstract syntax tree (AST).

Main component of the Xtext framework is a domain specific language for describing concrete syntax of a developed language. The described concrete syntax is mapped to an in-memory representation model, produced by a parser when an input file with the grammar specification is loaded. Xtext grammar includes mechanisms to combine a developed language with existing grammars of other languages, enabling grammar compositions. This mechanism is called *grammar mixin* [xte13].

Xtext parser creates in-memory object graphs, represented by EMF Ecore metamodel instances, while consuming the grammar specification. The parser can infer Ecore metamodels representing the abstract syntax of the language from a specified grammar, but also can generate a grammar from an imported Ecore metamodel.

Parsing of a specified grammar in Xtext can be separated into four phases: *lexing*, *parsing*, *linking*, and *validation*.

In the first phase, called *lexing*, a sequence of characters is transformed into a sequence of *tokens*, where token represents a strongly typed part of region of the input sequence. It consists of a number of characters, and is matched by a specific terminal rule or keyword, and represents an atomic symbol of the developed language.

Xtext comes with predefined grammars containing most used terminal rules for data types, such as strings and integers, which can be imported for reuse into the grammar of the developed language. An example of a terminal rule is presented in Listing 6.19.

Listing 6.19: An example of a terminal rule

```
1 terminal ID:
2 ('^')?('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Token ID from Listing 6.19 starts with an optional character '^', followed by a letter ('a'..'z'|'A'..'Z') or underscore ('\_'), followed by any number of letters, underscores, and numbers ('0'..'9'). Terminals are expressed using Extended Bacus-Naur Form-like rules, with four different possible cardinalities: exactly one (default), one or none (operator ?), zero or more (operator \*), and one or more (operator +).

Keywords are represented by terminal rule literals. Furthermore, to allow internal grammar composition, rules can refer to other rules, called *rule calls*. An example of a rule call is presented in Listing 6.20. Here, a rule named *QualifiedName* is specified as a combination of one or more IDs from Listing 6.19, concatenated with dot symbol.

Listing 6.20: An example of a rule call

```
1 QualifiedName:
2 ID ('.' ID)*
```

In the second phase, called *parsing*, a parser is fed with a sequence of terminals, on which parser rules are applied. Thereof, a tree of non-terminal and terminal tokens is produced

by the parser. This tree is composed of objects comprising the linked abstract syntax tree (AST).

There is a number of expressions for directing how the AST is constructed. *Assignments* are used for assigning the consumed information to a feature of currently produced object within a rule. There are three different assignment operators, each with a different semantics: the = operator used for single valued features, the += operator for multi valued features, and the ?= operator for boolean features.

One important feature of Xtext is *cross-linking* of produced elements from rules within a grammar definition. The syntax of cross-links is presented in Listing 6.21.

Listing 6.21: Xtext cross-link rule definition

```
1 '[' type=TypeRef ( '[' ^terminal = CrossReferenceableTerminal )? ' ] ';
```

The cross-linking feature of Xtext enables definition of which property of an Xtext element of the developed language grammar can be used for referencing to it from other elements in the grammar. As can be seen from Listing 6.21 there is an optional part of a cross link composed of a vertical bar followed by *CrossReferenceableTerminal*. This part describes the concrete text by which the cross link is established. If it is omitted, a rule with a name ID is used as text for establishing the cross link.

For instance, in our test language for referring to a specific test scenario in an *initialize* statement within a test case, the name of the test scenario is used for linking. Therefore, the cross referenceable terminal for a test scenario is its user defined name.

In the *linking* phase, the cross links explained earlier are resolved. Linking enables specification of references between elements of the language using so called cross links. Beside defining cross links within the grammar, it might be necessary to specify linking semantics, usually provided using the scoping API. Xtext uses lazy linking by default, which is a preferred use of linking due to improved performance, where it is not needed to load a complete model. However, cyclic linking is not supported in Xtext.

Scoping is used for determining which elements can be referenced from a certain link, based on the context in which the link is defined. While declaration of a cross link in grammar defines a type of objects which can be referenced from a certain link, filtering of objects of the specified type which can be referenced from a link is done by implementing so called scope providers. Scoping in Xtext is divided into global and local scoping. Global scopes, on one hand, define visibility of elements coming from an external resource, that can be referenced from a certain link. Local scoping, on the other hand, is used for restricting the visibility of elements referable from a certain link based on the context of its definition.

For instance, in our test language, scoping is used for constraining the set of activity nodes which can be declared in an order assertion. More precisely, only activity nodes which belong to the activity under test should be referable from an order assertion.

Similarly, scoping was used for constraining the set of referable activity parameters from a test case, for declaring the input to the activity under test.

The default linking in Xtext relies on a scope service, which provides context for each element in a program. During the cross reference resolution, the linker asks the scope service to provide the scope for the elements that are referable from the cross link it is trying to resolve, and if the element from the cross reference is in that scope, the cross reference is resolved by the linker. Otherwise, an error is presented in the editor.

Finally, in the *validation* phase, the abstract syntax tree can be semantically analyzed to check the overall correctness of the program. This analysis includes type checking and custom constraint checking concerning the semantics of the program elements.

For instance, in our test specification language, in an order assertion specification it is not allowed to define two adjacent escape characters (e.g., *nodeA*, \*, \*, *nodeB*, for more details on order assertions implementation cf. Chapter 7). Therefore, we have defined the validation rule that checks the specification of each order assertion in a test case, and each invalid order specification will be marked inside the editor during test creation. Implementing constraint checking such as this one, is less complex once the parsing is done and the abstract syntax tree is generated, rather than integrating it into the parsing phase.

Abstract syntax tree of a specified grammar in Xtext is represented by an Ecore metamodel generated from the grammar, or manually created and imported into the grammar. Complete definition of the language is contained within an instance of an EPackage element, containing a number of EClass instances for each parser rule, and a number of EDataType instances for each terminal rule or a data type rule defined within the grammar.

Xtext provides generic implementations for language infrastructure, and also uses code generation for some components. Among generated components are the parser, the serializer, the inferred Ecore metamodel, and other. For configuring the generator, Xtext uses a domain specific language called *modeling workflow engine* (MWE).

The general architecture of the language generator is composed of generator fragments, an URI pointing to the grammar, and the file extensions for the developed language. By using generator fragments, different components for generating different parts of the language infrastructure can be declared and invoked in the order in which they are declared. There are fragments for generating parsers, serializers, the EMF code, and other components. Further details on specifying a generator workflow using MWE can be found in [xte13].

Xtext provides many additional features, such as serialization, formatting and encoding, which are described in more details in [xte13]. Instructions on installing the testing framework and a complete overview of the test specification language grammar implementation in Xtext are given in appendices A and B, respectively.



# Test Interpreter

In this chapter we will present our test interpreter, used for evaluating the test cases on the execution trace of an activity under test. First we give an overview of the framework and the execution trace provided by the fUML virtual machine, containing the information about the executed activity nodes, as well as the execution states comprised of objects and links created and modified by the activity execution. Thereof, we describe in detail the implementation of evaluating order assertions, state assertions, and OCL expressions. Finally, we conclude the chapter with the presentation of the test results model produced by the test interpreter, representing the results of the evaluated test cases and revisit the ATM example

## 7.1 Overview

Each test case in the test specification language specifies an activity under test, and a set of values provided as input to each activity input parameter of the activity under test (as described in Chapter 6). The activity, along with the input values, is provided into the fUML virtual machine, which executes the activity under test. Thereof, an activity execution trace, is produced by the virtual machine. The test interpreter, takes as input the specified test cases and the execution trace, performs the evaluation of each assertion within a test case, and produces the test results. The process of executing and evaluating test cases is shown in Figure 5.4.

The standardized fUML virtual machine does not provide means for implementing analysis methods, such as debugging and testing, as it provides means only for executing a specified fUML activity and records the output values of the activity parameters after the execution has been completed. For implementing analysis methods, such as debugging and testing, there are several important characteristics that have to be provided by the execution environment. One of those characteristics is *observability* [May14]. Observability of an execution environment presumes ability to observe the execution of a model during

runtime. Another important characteristic of an execution environment is *controllability*, that is ability to control executions being carried out by the execution environment. Finally, an important characteristic of an execution environment for analysis methods, such as testing, is *analyzability*. Analyzability presumes ability to analyze ongoing or completed executions based on the captured runtime execution.

Therefore, our testing framework is based on an implementation of extended version of the fUML virtual machine [May14]. This extended virtual machine contains (i) an event mechanism enabling the runtime observation of model executions carried out by the fUML virtual machine, (ii) a command interface providing execution control over model executions carried out by the fUML virtual machine, and (iii) a trace model recorded for model executions carried out by the fUML virtual machine containing information regarding executed activity nodes, input and output relations between nodes and other. Test interpreter of our testing framework depends on this recorded trace model for evaluation of specified test case assertions.

As the first step, specified data in the test scenarios has to be translated into the fUML representation, in order to be provided to the virtual machine which executes the activity under test with the given input. As described in Chapter 6, a test scenario is composed of objects and links which can be provided as input or an initial state to the activity under test. Each object and link specified in a test scenario, and provided as input to the activity under test, is translated into the fUML representation. Furthermore, if a test scenario is specified as an initial state of the activity under test, all objects and links defined within it are translated into fUML representation and loaded as the initial state of the activity under test. In order to isolate specified test cases within a test suite, each time a test case is to be executed, the final state of the system is cleared and the initial state defined by the activity inputs or a test scenario set as initial state of the activity under test, is loaded into the virtual machine.

## 7.2 Trace Model

In Figure 7.1 a model of the execution trace, produced by the fUML virtual machine is presented. Main element of the trace model is the *Trace* class. It is composed of value instances (i.e., objects and links) created or modified during the execution of an activity under test, and activity execution objects containing information about each executed activity.

For each object or link, created or modified during execution of an activity, an instance of the *ValueInstance* class is created. Objects that were provided to the activity existed in the initial state, and thus they are referenced from the trace as initial locus value instances (the reference *initialLocusValueInstances*).

Whenever an object is instantiated or modified, a snapshot of that object containing the current values of its attributes is created, represented by the class *ValueSnapshot*. Each value instance contains all snapshots of that instance that were created during an

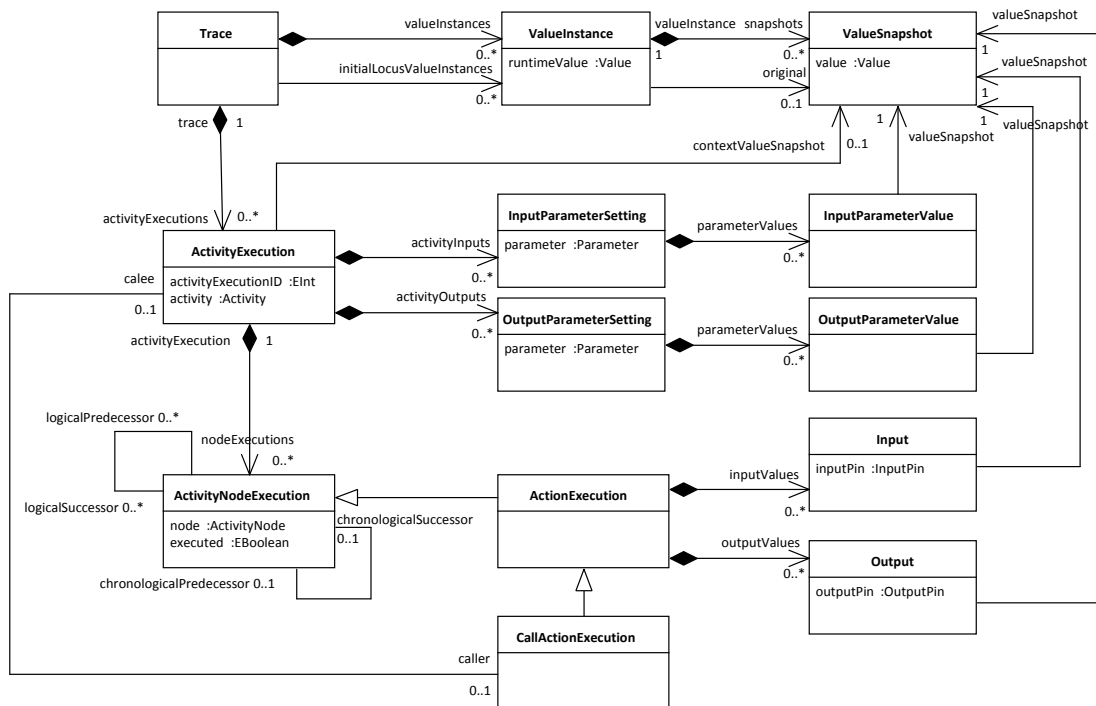


Figure 7.1: Excerpt of the execution trace model [May14]

activity execution (the reference *snapshots*), as well as a reference to the original value snapshot provided to or created during an activity execution (the reference *original*).

For each executed activity, an instance of the class *ActivityExecution* is created. Each activity execution contains an identifier (the attribute *activityExecutionID*) and a reference to the activity that was executed (the attribute *activity*). An object provided as context of an executed activity is referenced from an instance of corresponding activity execution (the reference *contextValueSnapshot*). For each input and output object provided to (from) an activity, a corresponding instance of *InputParameterSetting* and *OutputParameterSetting* is created, referring to the parameter for which it was created. Parameter settings refer to the value snapshots consumed or produced by the executed activity (the reference *valueSnapshot*).

For each node within an executed activity, an instance of the class *ActivityNodeExecution* is created. Activity node execution refers to the activity node for which it was created (the attribute *node*), and a flag value indicating if the node was executed or not (the attribute *executed*).

The chronological order of nodes within the executed activity, describing the time order in which the nodes were executed, is recorded by a unary association (association ends *chronologicalSuccessor* and *chronologicalPredecessor*). The logical dependency of nodes within the executed activity, describing which nodes provided input to which other

nodes, is recorded by another unary association (association ends *logicalSuccessor* and *logicalPredecessor*).

For each action within the executed activity, an instance of the *ActionExecution* is created. An action execution contains references to the input and output values provided to or from an executed action. If an action within the executed activity is either a call behavior action or a call operation action, for such action an instance of *CallActionExecution*, containing a reference to the executed activity, is created.

Based on this recorded trace of an activity execution, test interpreter is able to evaluate specified order and state assertions from a test suite. Implementation details of the test interpreter for each kind of assertion from the test specification language are presented in the rest of the chapter.

### 7.3 Order Assertions

Concurrency in an activity leads to the existence of a potentially large number of possible execution paths of that activity, which have to be considered in the test evaluation. In particular, order assertions checking the correct execution order of activity nodes have to be evaluated for each possible execution path of the activity under test. We have implemented two algorithms for evaluating order assertions, presented in the following subsections.

#### 7.3.1 Execution Tree Generation Algorithm

Once an activity is executed, the execution trace of that activity is recorded by the fUML virtual machine. The metamodel of the trace is presented in Figure 7.1. The execution trace is obtained from a single execution of the activity under test, for the given input defined by the test case being evaluated.

The information needed for evaluating order assertions consists of the logical input/output dependencies between the executed activity nodes captured by the execution trace. Thereby, an activity node *B* depends on an activity node *A*, if *B* received an object token or control token from *A* as input. This information is captured in the execution trace by links called *logicalSuccessor* and *logicalPredecessor* pointing from *A* to *B* and from *B* to *A*, respectively. Based on this logical order it is possible to compute execution paths of an activity for a given input.

An execution tree, generated by our algorithm and from which possible execution paths of an activity are computed, will be presented on an example activity depicted in Figure 7.2.

This activity is composed of eleven activity nodes. There are several important properties of the depicted example activity. The activity contains two potential starting nodes. The execution of the activity can start either from node *initial* or node *actionB*.



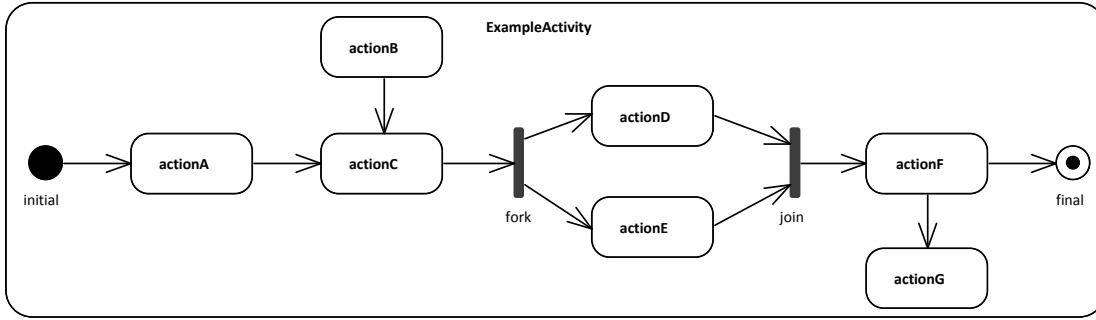


Figure 7.2: An example activity with several starting and several ending nodes

Furthermore, it contains two ending nodes, i.e., *final* node and *actionG* node. An important thing to notice about these two nodes is that, if the *final* node is executed before *actionG* node, the *actionG* node will not be executed. As the type of the *final* node is *activity final node*, once it is executed, the execution of the whole activity is terminated by the virtual machine (cf. Figure 4.2). However, if *actionG* node is executed first, the *final* node will be executed also.

Finally, the activity contains a parallel path composed of two nodes, i.e., *actionD* and *actionE*. As these two actions lay on a parallel path, they can be executed in any order. An execution trace of the activity from Figure 7.2, produced by the virtual machine, is presented in Figure 7.3.

As can be seen in Figure 7.3, for each node in the activity, a corresponding instance of an *activity node execution* or an *action execution* is created. The logical dependencies between nodes, that is which node provided input to, or consumed output from, which other node is represented by the links *logicalSucc.* and *logicalPred.*, respectively. Nodes without any incoming *logicalSucc.* links are called *starting nodes*, and nodes without any outgoing *logicalSucc.* links are called *ending nodes*.

---

**Algorithm 1** Generation of execution trees for each starting node from a given activity execution  $e$

---

```

1: procedure GENERATE TREES(ActivityExecution e)
2:   for  $n$  in  $e.nodeExecutions$  do
3:     if  $predecessors(n).size() = 0$  then
4:        $list \leftarrow GenerateTree(n)$ 
5:   return  $list$ 

```

---

The entry point of our execution tree generation procedure is presented in Algorithm 1. In this procedure, for each node of the activity from the execution trace, we check if that node has no predecessors, i.e., it is a *starting node*, and if so we invoke a procedure *GenerateTree* depicted in Algorithm 2 on it, in order to generate an execution tree of that node. Finally, we add it to a list of generated execution tree nodes, returned by the

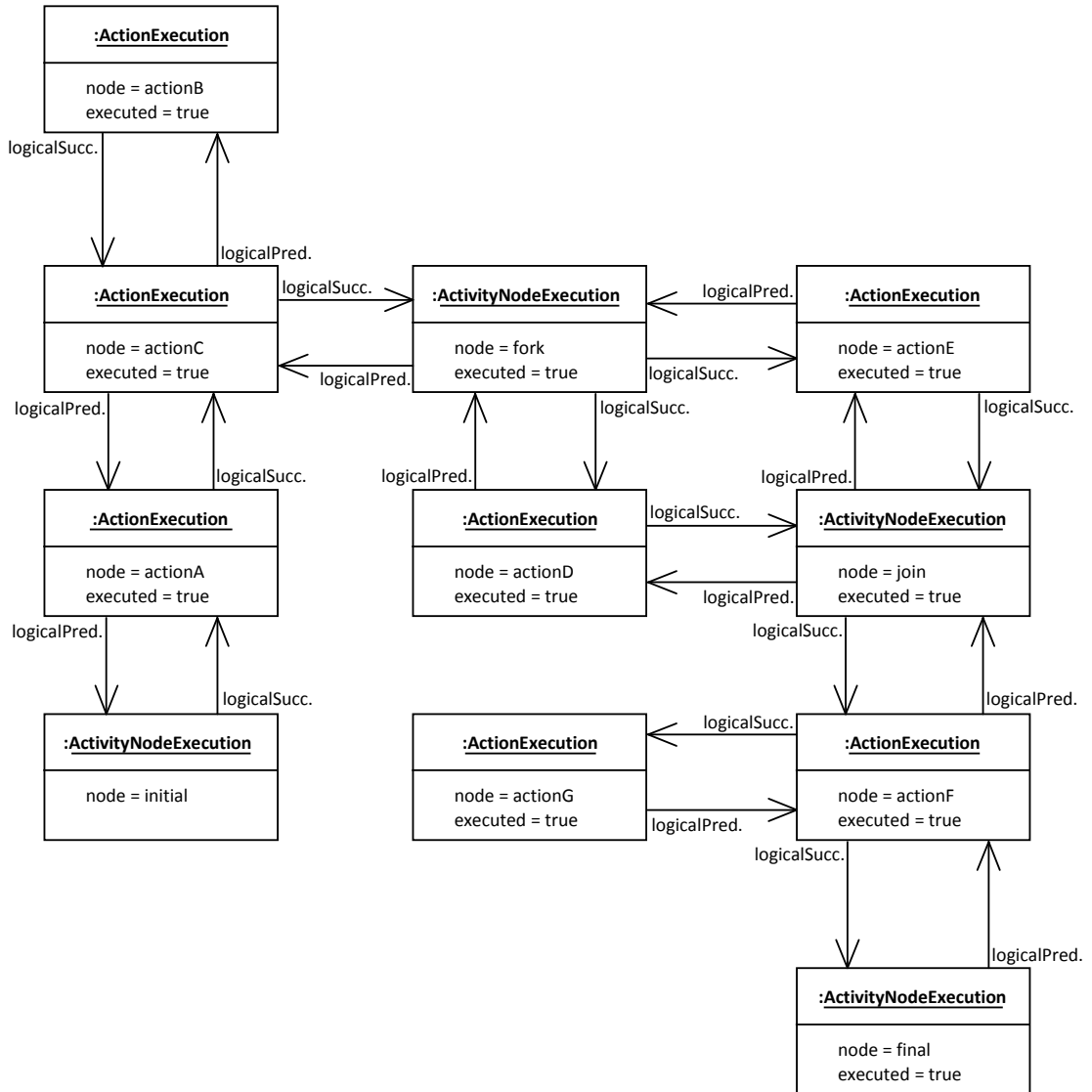


Figure 7.3: An excerpt of an execution trace produced by the fUML virtual machine for the activity from Figure 7.2

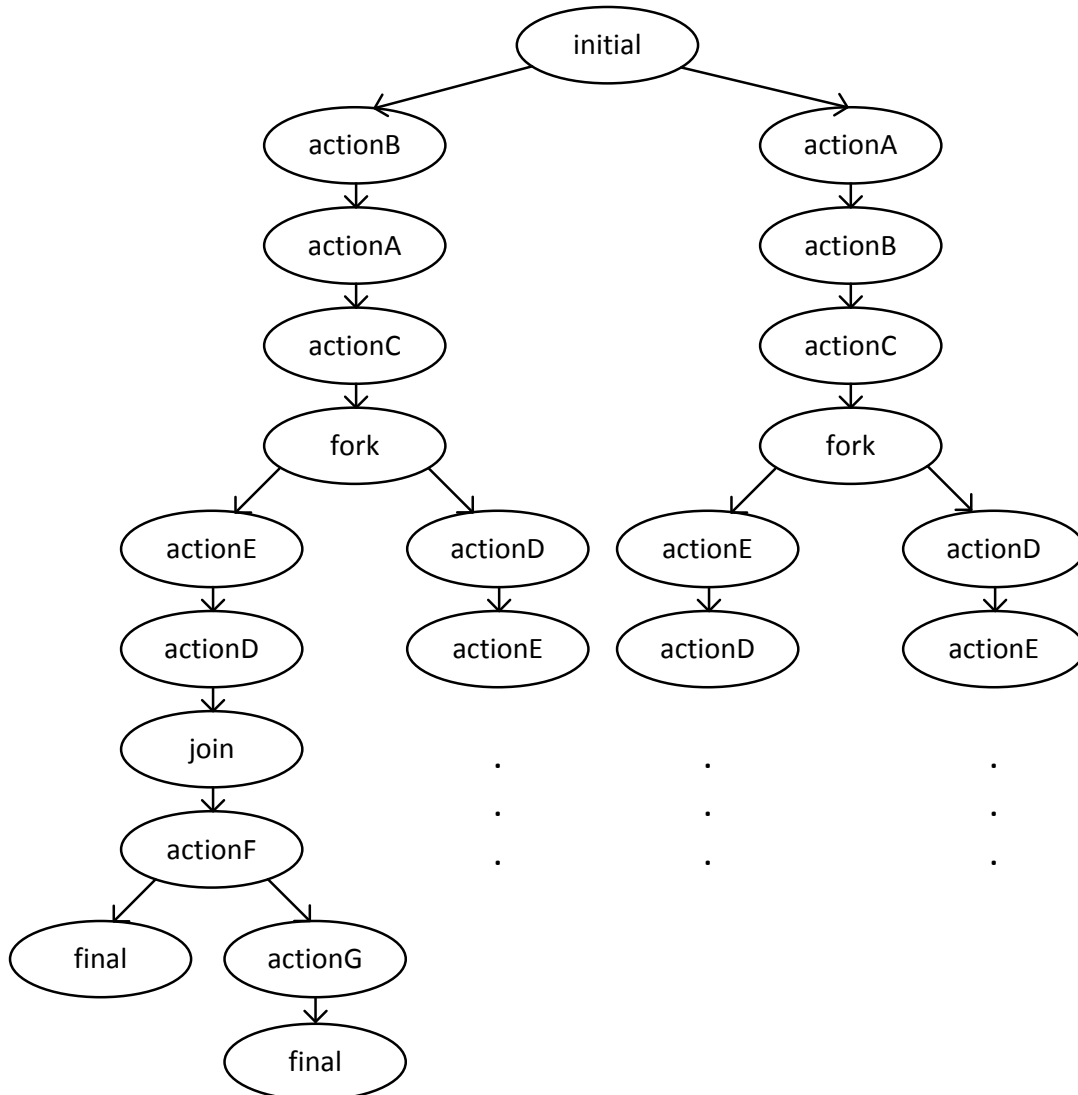


Figure 7.4: Execution tree generated from the execution trace from Figure 7.3, with the node *initial* as its root

procedure. In each generated execution tree, each branch from the root node to each leaf node, represents an execution path of the activity. An execution tree, starting from node *initial*, generated from the execution trace of the example activity is depicted in Figure 7.4.

The execution tree of the example activity is generated in following order of steps. Once the *initial* node is executed, the execution can proceed with either the node *actionA* or *actionB*. Therefore, two execution tree nodes referring to the *actionA* and *actionB* nodes, are created and added as children of the execution tree node referring to the *initial* node.

Assuming that the next chosen node to be executed is the node *actionA*, an execution tree node referring to it is created and added as a child of the previously created execution tree node referring to the node *initial*. As the node *actionC* which received the input from the node *actionA* still awaits input from the node *actionB*, it cannot be executed. At the same time, the node *actionB* has no incoming edges, and therefore is the only node which can be executed at that point in time. Therefore, an execution tree node referring to the node *actionB* is created and added as a child of the previously created execution tree node referring to the node *actionA*.

Once the node *actionB* is executed, the node *actionC* becomes executable, and since it is the only node that is executable at that point in time, it is the next chosen node. Execution of node *actionC* is followed by execution of the node *fork*, after which possible executable nodes are *actionD* and *actionE*. The appropriate execution tree nodes are created and added into the execution tree in the same way as in the previous steps, and this process continues until a node which has no outgoing edges, and therewith no successor nodes, is reached. The remainder of each execution path after execution of the parallel nodes *actionD* and *actionE* is same, therefore it is only presented once in Figure 7.4.

The procedure for generating an execution tree for a given activity node execution is presented in Algorithm 2. For a given activity node execution  $n$  from the trace, we create an instance of the execution tree node  $e$ . An execution tree node contains reference to the activity node execution of the execution trace for which it was created (node  $n$  itself), and a list of its children, i.e., nodes which may follow the activity node execution  $n$  on an execution path of the activity under test.

Thereof, all potential successor nodes of the current node  $n$  are computed, i.e., operation  $potentialSuccessors(n)$ . Potential successor nodes are all nodes from the trace that might, under certain conditions follow the current node on an execution path of the activity under test.

For each node  $m$  in the list of potential successors, several conditions defining whether the node is executable in the context of the current path, implemented by the operation  $isExecutable(m, n)$ , are checked and if they are fulfilled, an execution tree node for  $m$  is created and added as a child of the current node in the execution tree. Finally, this process is recursively repeated for each node executable in context of an execution path, creating a corresponding execution tree node for it, and adding the generated result as a child of the current execution tree node.

The procedure for retrieving all potential successor nodes of an execution tree node, used in line 3 in Algorithm 2, is presented in Algorithm 3. When computing all potential successors of a given node  $n$  from the trace, first we have to check all of the starting nodes, that is those nodes which do not contain any logical predecessors in the trace. Each such starting node can be added to the list of all potential successors of the current node (line 6 in Algorithm 3). For instance, in the activity depicted in Figure 7.2, when computing the potential successors of the *initial* node, this step will ensure that the path

with the node *actionB* will be included.

---

**Algorithm 2** Generation of an execution tree for a given activity node execution *n*

---

```

1: procedure GENERATETREE(ActivityNodeExecution n)
2:   e ← new ExecutionTreeNode(n)
3:   list ← potentialSuccessors(n)
4:   for m in list do
5:     if isExecutable(m, n) then
6:       e.children ← GenerateTree(m)
7:   return e

```

---



---

**Algorithm 3** Retrieve potential successors of a given activity node execution *n*

---

```

1: procedure POTENTIALSUCCESSORS(ActivityNodeExecution n)
2:   psList ← new list
3:   snList ← startNodes()
4:   for s in snList do
5:     if s ≠ n then
6:       psList ← s
7:       psList ← descendants(s) / descendants(n) / n
8:   psList ← successors(n)
9:   aList ← ancestors(n)
10:  for a in aList do
11:    dList ← descendants(a)
12:    for d in dList do
13:      if d ≠ n and isNotAncestor(s, n) then
14:        psList ← d
return psList

```

---

For the purpose of explaining further steps of the algorithm, we will call the execution path on which the node *n* lies the *main path*. Each *descendant* of each starting node from the trace, up to the node *n*, can be added to the list of all potential successors of the node *n*. A *descendant* of a node *x* is any direct or indirect logical successor of the node *x*. Therefore, all descendants of each starting node, which are not descendants of the node *n* and the node *n* itself, can be added to the list of potential successors of the current node *n* (line 7 in Algorithm 3). The operator '/' is used here as a relative complement, taken from set theory. For instance, A/B is a subset of A such that each element in this subset belongs to A and at the same time does not belong to B. This step ensures that any node laying on a parallel path that did not originate from the *main path* will be included in the list of potential successors for the node *n*.

For instance, in the activity depicted in Figure 7.2, when computing potential successors of the *initial* node, this step would ensure that any node laying on a path between *actionB* and *actionC* would be included in a list of potential successors of the *initial* node.

Additionally, each direct logical successor of the node  $n$  is a potential successor of the node  $n$ , so it can be automatically added to the list of all potential successor nodes of the node  $n$  (line 8 in Algorithm 3).

Finally, each descendant of each *ancestor* of the node  $n$ , which is not an ancestor of the node  $n$ , can be added to the list of potential successors of the node  $n$ . An *ancestor* of a node  $x$  is any direct or indirect logical predecessor of the node  $x$ . This step ensures that that any node laying on a parallel path that forked from the same path as the main path will be included as a potential successor of the node  $n$  (lines 12-14 in Algorithm 3). For instance, in the activity depicted in Figure 7.2, when computing potential successors of the execution tree node of the *actionE*, this step ensures that the node *actionD* will be included.

The procedure for checking if a node  $m$  from the list of potential successors of the node  $n$  can be added as a successor of the node  $n$  on a certain execution path, used in line 4 in Algorithm 2, is presented in Algorithm 4. A node  $m$  can be added to the list of successor nodes of the given node  $n$  if it was executed, if it was not already added on the current path up to the node  $n$ , and if all logical predecessors of such node have been already added on the current path to the node  $n$ .

---

**Algorithm 4** Check if a potential successor of a given activity node execution  $m$  can be added as a child of a given activity node execution  $n$

---

```
1: procedure ISEXECUTABLE(ActivityNodeExecution m, ActivityNodeExecution n)
2:   if isAlreadyAdded( $m, n$ ) or isExecuted( $m$ ) = false then
3:     return false
4:    $list \leftarrow predecessors(m)$ 
5:   for  $p$  in  $list$  do
6:     if isAlreadyAdded( $p, n$ ) then
7:       return false
8:   return true
```

---

Once the execution trees for each starting node from the trace are generated, by applying a simple depth first search, we can go through and validate each execution path of an activity under test. Union of all paths between the root node and the leaf nodes from each generated execution tree represents a set of all activity execution paths.

An important disadvantage of the execution tree generation algorithm is that it can lead to state explosion, and generating all of the paths for an order assertion might be inefficient for very loosely coupled activities (activities with a large number of starting or ending nodes and parallel constructs such as fork and join). Therefore, we have implemented an additional more efficient algorithm, based on creating and analyzing an adjacency matrix representation of an execution trace, presented in the following subsection.

Validation of an order assertion for an activity under test can be performed either by

	ini.	a	b	c	fk.	d	e	jn.	f	g	fin.
initial (ini.)		T									
actionA (a)				T							
actionB (b)				T							
actionC (c)					T						
fork (fk.)						T	T				
actionD (d)								T			
actionE (e)								T			
join (jn.)									T		
actionF (f)										T	T
actionG (g)											
final (fin.)											

Figure 7.5: Adjacency matrix of the activity from Figure 7.2, generated from the execution trace from Figure 7.3 produced by fUML virtual machine

analyzing an execution path obtained from an execution tree of the activity, or by analyzing the adjacency matrix (as described in the following subsection). The algorithm for validating an order assertion against an execution path or the adjacency matrix will be presented in the following subsection.

### 7.3.2 Adjacency Matrix Analysis Algorithm

We have implemented an algorithm based on adjacency matrix to evaluate order assertions in the presence of concurrency, that enables to verify the correctness of the execution order of activity nodes for a given input. As a first step the algorithm transforms the execution trace of an activity under test into an adjacency matrix [CSRL01].

The adjacency matrix constructed for the execution trace from Figure 7.3 is presented in Figure 7.5. The matrix constitutes a two dimensional array of Boolean values, where a true value (abbreviated with *T*) indicates the existence of a *logicalSucc.* link between two activity nodes. For instance, a true value in the first row and the second column indicates the mentioned *logicalSucc.* link pointing from *initial* node to the *actionA* node.

Based on the constructed adjacency matrix, order assertions can be evaluated efficiently by analyzing the dependencies between activity nodes specified in the order assertions. For instance, to evaluate an order assertion **assertOrder** \*, *A*, *B*, \*, we have to check whether *B* depends on *A*, i.e., whether a true value in the adjacency matrix indicates *B* as being adjacent to *A*. If this is not the case, there exists no input/output dependency between *A* and *B*, and hence they may be executed in reverse order.

Furthermore, we have to check that there are no other nodes independent of both *A* and *B*, i.e., nodes that lie on parallel paths and may be executed between them. We can

compute all ancestors and all descendants for both node  $A$  and node  $B$ , and check if there is a node  $X$  which does not belong to either ancestors or descendants of either the node  $A$  or the node  $B$ . If there is at least one such node  $X$ , then the order assertion should fail.

For the evaluation of jokers ' $\_$ ' and '\*', also indirect input/output dependencies between activity nodes have to be considered, which can also be efficiently calculated from the adjacency matrix. For instance, to evaluate an order assertion **assertOrder**  $A, \_, B$ , we have to check whether an arbitrary activity node  $X$  exists on which  $B$  depends and which itself depends on  $A$ , i.e.,  $X$  provided input to  $B$  and received input from  $A$ . Additionally, we need to check that there are no nodes independent of  $A$ ,  $X$ , and  $B$ , which lie on parallel branches.

In the evaluation of order assertions, our algorithm always considers groups of three activity node specifications. For instance, an order assertion **assertOrder**  $A, B, C, D$  is divided into two groups  $\{A, B, C\}$  and  $\{B, C, D\}$ . The evaluation result of an order assertion is then the conjunction of the evaluation results for each group.

In order to make the order assertion evaluation algorithm feasible, we constrain the order assertion specification so that no subsequent use of jokers is allowed, i.e.,  $(*, *)$ ,  $(*, \_)$ ,  $(\_, *)$ , and  $(\_, \_)$ . This leads to existence of thirteen patterns for each group of three node specifications (e.g.,  $node, *, node$ ).

Therefore, we implemented a set of rules for each pattern of a three node group, such as the ones presented earlier, and validate them against each group from the order specification, either against the adjacency matrix, or a single execution path computed from the execution tree presented in previous subsection. The order assertion can be evaluated first on the adjacency matrix, to validate the execution order of any possible path of the activity under test in an efficient way. In case the validation against the adjacency matrix fails, we perform a search through the execution tree, in order to find one or more counter examples of execution paths which violate the order specification. As soon as one or more incorrect paths are found, the exploration of the generated execution tree is terminated, and the found counter example presented to the user.

## 7.4 State Assertions

As described in Chapter 6, a test case may contain state assertions for evaluating state of execution of an activity under test. Each state assertion is composed of definition of a time frame selecting a set of activity execution states to be evaluated, and a set of state expressions evaluated on the objects and links in the selected set of states.

Therefore, evaluation of state assertions in the testing framework is divided into two phases. In the first phase, based on the specified time frame (cf. Section 6.1.5) a set of relevant snapshots of objects and links from the execution trace, comprising a set of execution states of the activity under test, are collected.

As described in Section 6.1.5, for specifying a time frame of a state assertion it is possible to refer to either actions within the activity under test, or by specifying OCL constraints



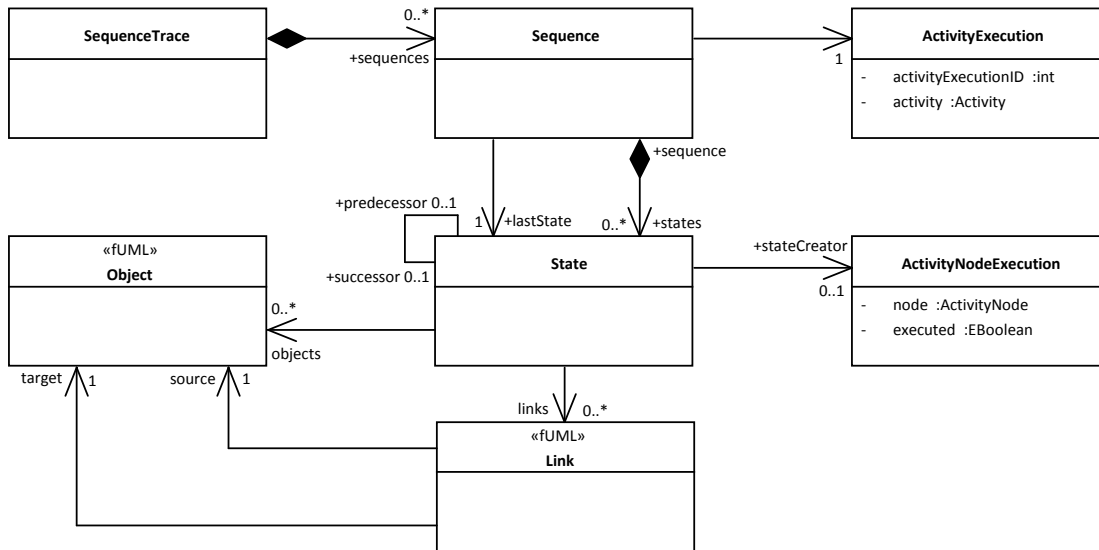


Figure 7.6: Excerpt of sequence execution trace model

which determine start and end state of the time frame. In order to enable selection of relevant states based on the specified time frame, we convert the trace model (described by the model from Figure 7.1) into a so called sequence execution trace, depicted in Figure 7.6.

Looking at the original execution trace in Figure 7.1, information for both computing the selected set of objects and links defined by time frames and for evaluating the state expressions within a state assertion is sufficient. However, computing the time frames defined within a state assertion, especially in case the time frame is defined using OCL, as well as evaluating OCL expressions for each execution state in the time frame, by using the execution trace from Figure 7.1 directly, is not straightforward. For instance, every time an OCL constraint is to be evaluated in a specified time frame, a set of objects and links comprising the relevant states for the time frame has to be computed from the execution trace. Therefore, we have created the sequence trace representation described in the following, and a transformation of execution trace into this representation, with the aim to reduce the complexity of implementing the state assertion evaluation and the use of OCL constraints.

A sequence trace (the class *SequenceTrace*) is composed of a number of sequences (class *Sequence* in Figure 7.6) created for each activity execution from the original execution trace (class *ActivityExecution* from Figure 7.1). A sequence is composed of a set of states (class *State* in Figure 7.6), created for each action execution from the original trace, which modified the state of the system in some way. More precisely, for each action (*ActionExecution* class from Figure 7.1) whose execution modifies the state of the system in some way, a new state (*State* class from Figure 7.6) is created. There is a set of action types which can modify the system state, such as *CreateObjectAction* or

*DestroyObjectAction*, for which the transformation creates a new state, adding a reference to a new snapshot or removing a reference to an existing snapshot from the previous state in order to create the new state, and setting the respective action as the creator of the new state (link *stateCreator* in Figure 7.6). States are composed of objects and links created or modified by the creator action of the corresponding state, and those objects and links from the previous state which were unaffected by the creator action of the new state. This way, we have a distinct set of states with objects and links existing at the specific point in time of execution, representing state transitions of the system under test. Furthermore, the chronological order between states is recorded (association ends *successor* and *predecessor* in Figure 7.6). Again, as described earlier, having the states created before the actual evaluation of the state expressions and OCL constraints, rather than fetching this information directly from the execution trace produced by the virtual machine during test evaluation, makes implementation of the evaluation less complex and more efficient.

Once the sequence trace is generated, the evaluation of the time frames of state assertions is performed in the following way. If the start or end point of the time frame is specified by referring to actions within the activity under test (cf. Figure 6.1), then the set of states which are to be selected for evaluation is composed of the state created by the action associated with the *after* operator and all subsequent states until the last state before the execution of the action associated with the *until* operator.

On the other hand, if the start and end points of the time frame are specified by using OCL expressions, then the set of states which are to be selected for evaluation is composed of those states that exist after the first state in the state sequence in which the specified OCL constraint for the start point evaluates to true, and all subsequent states until the state in which the specified OCL constraint for the end point evaluates to true for the first time in the state sequence.

In the second phase, state expressions defined within the state assertion are evaluated on the selected set of states. As described in Section 6.1.5, a state expression might be specified for a single property or the whole object provided as input or output of an action within the activity under test. In case a state expressions for a single property of an object is specified, the object snapshots are retrieved from each state from a selected set of states, and the value of the specified property of each snapshot is asserted against the specified expected value. In case a state expression for the whole object is specified, each property value of relevant snapshots, retrieved from the selected set of states, is compared to a corresponding property value of the specified expected object from a test scenario.

The real advantage of the sequence trace, compare to using the execution trace directly, comes with OCL constraint evaluation. If an OCL expression was specified in the body of a state assertion, the OCL constraint is evaluated against each state from the selected set of states by the time frame. As opposed to the state expressions from the test language, which are specified for a single object produced or modified during the execution of an activity under test, the OCL constraints are usually specified as a property of the whole

system state at certain point in time of its execution. Therefore, it usually includes computing all existing object and link snapshots comprising the state under evaluation, and validation of the specified OCL constraint on the state. Having the sequence trace computed before the actual evaluation of the OCL constraints, simplifies the evaluation process. The implementation of evaluating the OCL expressions in both the time frames, as well as in the body of a state assertion, is described in the following section.

## 7.5 OCL Expressions

For integrating OCL expression language with our testing framework we have used the DresdenOCL API (application programming interface) [FJS<sup>+</sup>11]. DresdenOCL provides an integration process of OCL with different modeling languages based on EMF framework, built around small specifications out of which all necessary artifacts for editing and evaluating OCL expressions can be created.

The integration process is composed of five phases. During the first phase, called *metamodel integration*, the metamodels of OCL and the modeling language are combined. The resulting metamodel is used by EMF code generator to generate Java classes for the resulting metamodel. In the following phase, called *concrete syntax integration*, the textual syntax of both languages is integrated and used for the generation of a textual parser and editor. The first two phases are only required for embedding OCL definitions into the modeling language.

As we are using the external integration of the OCL with our test specification language, that is, the OCL expressions are specified outside of the test specification language, these two steps are not used. Third step is *metamodel adaptation*, required for both internal and external integration, and consists of creation of a pivot model representation of concepts within the modeling language, enabling parsing of OCL constraints that refer to the elements of the language.

Fourth step, called *static semantics integration*, results in static semantics analysis for integrated languages from step two. Fourth step is optional, as the metamodel adaptation from step three is sufficient for external OCL definitions, to allow semantic analysis of constraints. In the last step, called *dynamic semantics integration*, infrastructure for evaluation of integrated OCL constraints is provided. The excerpt of the package architecture<sup>1</sup> of DresdenOCL API is depicted in Figure 7.7.

As can be seen in Figure 7.7, at the top of the DresdenOCL architecture is an API providing a facade for access to the DresdenOCL tools, such as OCL parser and interpreter, code generation and model browsing, from other plugins within the EMF framework. On the next level below the API are the tools, such as OCL parser and OCL interpreter, as depicted in the figure.

Below the tools level is the OCL level. This level consists of libraries describing the OCL syntax and semantics for use by the tools at the higher level. As depicted in

<sup>1</sup>taken from <http://emftext.org/index.php/DresdenOCL:Documentation>

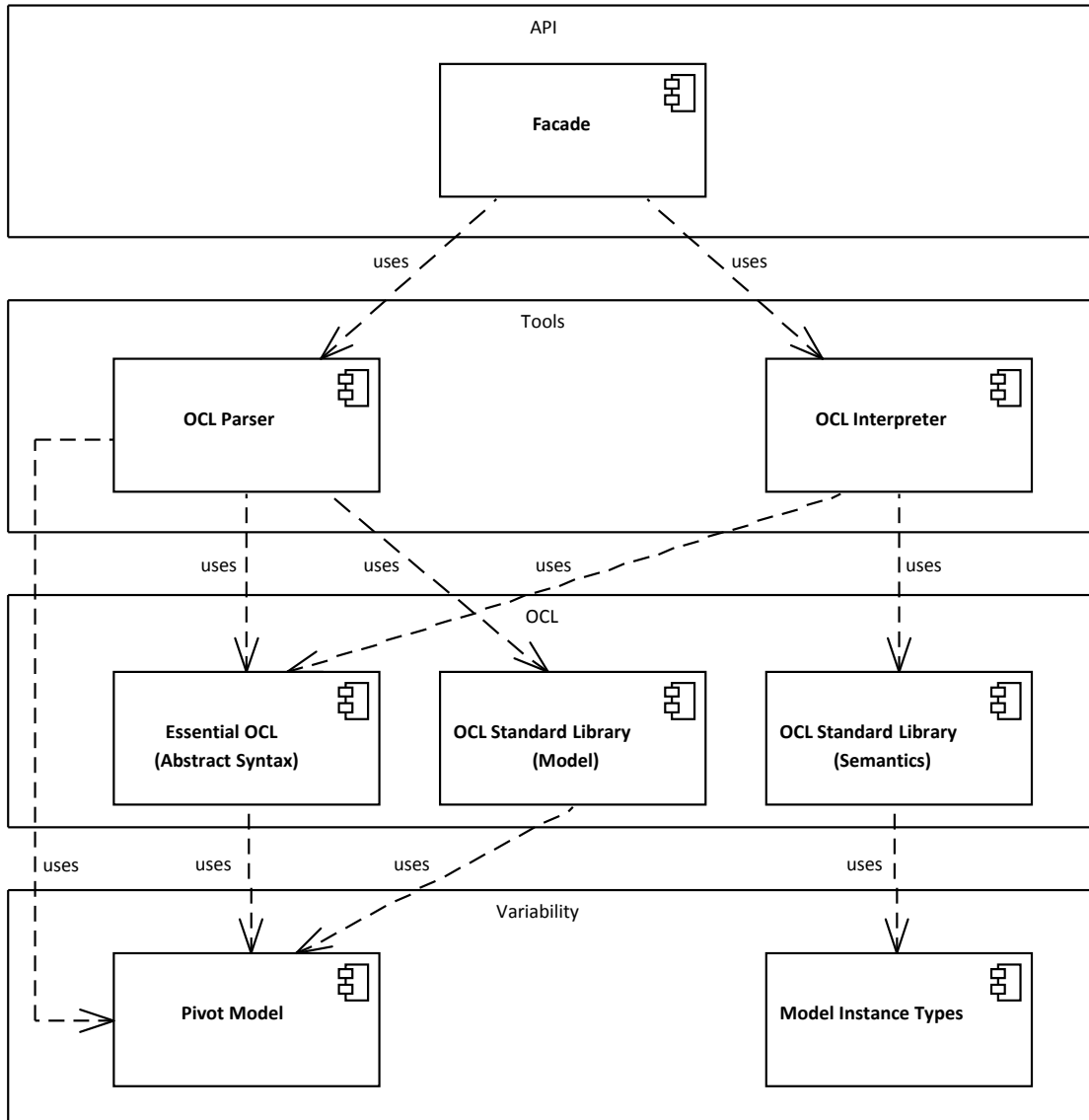


Figure 7.7: Excerpt of the package architecture of DresdenOCL

Figure 7.7, there are three packages defined at this level. The *Essential OCL* package contains definition of the abstract syntax of the OCL necessary for both interpretation and evaluation of the OCL expressions by the parser and interpreter at the tools level. The *OCL Standard Library Model* package contains models of all standard operations which can be used in OCL expressions, as defined by the OCL standard (cf. [Obj12]). The last package *OCL Standard Library Semantics* contains implementation of the operations from the *OCL Standard Library Model* package, used by the OCL interpreter at the tools level.

Finally, at the *variability* level of the DresdenOCL architecture, the *Pivot* model and the *Instance Types* model are defined. The pivot model is composed of adapter types for each type from the metamodel of the language. It is used to abstract from the original metamodel of the language on which the OCL constraints are defined, and is used during the OCL constraint parsing phase. The instance types model is used for adapting the model instance elements, used during the interpretation of the OCL constraints on a model instance.

### 7.5.1 Metamodel Adaptation and OCL Interpretation

During the metamodel adaptation phase, for each class from the metamodel of fUML a corresponding adapter class is generated. The fUML metamodel and an instance of it is represented by corresponding interfaces in the DresdenAPI, and can be retrieved by using a so called model and instance provider utility classes, respectively. These two providers are created by DresdenOCL during the generation of the metamodel adapters.

For example, an fUML model is represented by an object of *FUMLModel* class, generated by the DresdenOCL framework during model adaptation. Furthermore, for each concept from the fUML metamodel, such as for example *Class\_* or *Association* concepts, a corresponding adaptation class (i.e., *FUMLClass* and *FUMLAssociation*, respectively) is generated.

By using a facade utility class from the API, first a metamodel adapter is retrieved, containing a model instance provider utility. The model instance provider can be used for loading an existing instance of the adapted metamodel, or for creating it during run-time.

Model instance provider makes use of several special classes, most notably *FUMLModelInstance* and *FUMLModelInstanceObject*, representing the instances of an fUML model and its elements, respectively. For creating and populating an instance of an fUML model, the class *FUMLModelInstanceFactory* is used. This class provides functionality for creating instances of fUML models, instances of model elements such as objects and links, and instances of primitive types used within the model. Finally, the *FUMLModelInstanceProvider* class can be used for creating an empty instance of fUML model adaptation, or loading a predefined instance from a resource such as an XML serialization of an fUML model.

At the beginning of evaluation of an OCL constraint, based on the specified time frame from the state assertion, a corresponding state is retrieved from the sequence trace (cf.

Figure 7.6). For this state, a corresponding adapter instance, representing a single system state, is created. This model instance adapter is populated with adapter instances for each object and link from the original state retrieved from sequence trace. Once the model instance of a state is created, it is stored so it can be reused for evaluation of other OCL constraints in the test suite.

After creating the model instance of a state, the evaluation of the OCL constraint itself can be performed. For this step an OCL interpreter instance is initialized for a given model instance representing the state, by using the generated adaptation API. Finally, the OCL interpreter takes as input the OCL constraint to be evaluated, and the model instance representing the state in which the OCL constraint is to be evaluated, and finally returns a boolean value indicating the result of the OCL constraint evaluation.

In case a context object was defined for the OCL constraint (cf. Section 6.1.5), the object is retrieved from the model instance representing the state and the OCL constraint is evaluated on it. However, if a context object was not defined, then the OCL constraint is evaluated for the complete model instance representing the state (cf. Section 6.1.5).

As described in the previous chapter, an OCL constraint can be used in a state assertion to define a start or end point of a time frame. In this case, the specified OCL constraint is evaluated on each model instance adapter of each state from the sequence trace in a chronological order, and the first state in which the constraint is evaluated to true is taken as the start or end point of the specified time frame.

For evaluating the OCL expressions, each primitive operation, such as for instance summation of numbers, has to be implemented for each primitive type supported in the adapted modeling language. As we are defining test cases for fUML models, the adaptation is done for each primitive type in fUML, such as Integer, String, and Boolean. For instance, for the string type operations such as string concatenation or extraction of a substring, an appropriate implementation has to be provided in our testing framework. Further details on the architecture of the DresdenOCL and implementation of the OCL parser and interpreter are given on the framework website<sup>2</sup>.

## 7.6 Test Results

In order to provide useful feedback to the user of the testing framework, we have implemented a model of the test results produced from an execution of a specified test suite. The model is presented in Figure 7.8. In this figure, the concepts from the testing language are filled with dots, while the fUML concepts are grayed out, in order to distinguish them from the concepts of the results model.

Main component of the test results model is the *TestSuiteResult*. It is created for an executed test suite, and contains information about outcome of each defined test case within it (association *testCaseResults*). For each test case from an executed test suite, an

---

<sup>2</sup><http://emftext.org/index.php/DresdenOCL>

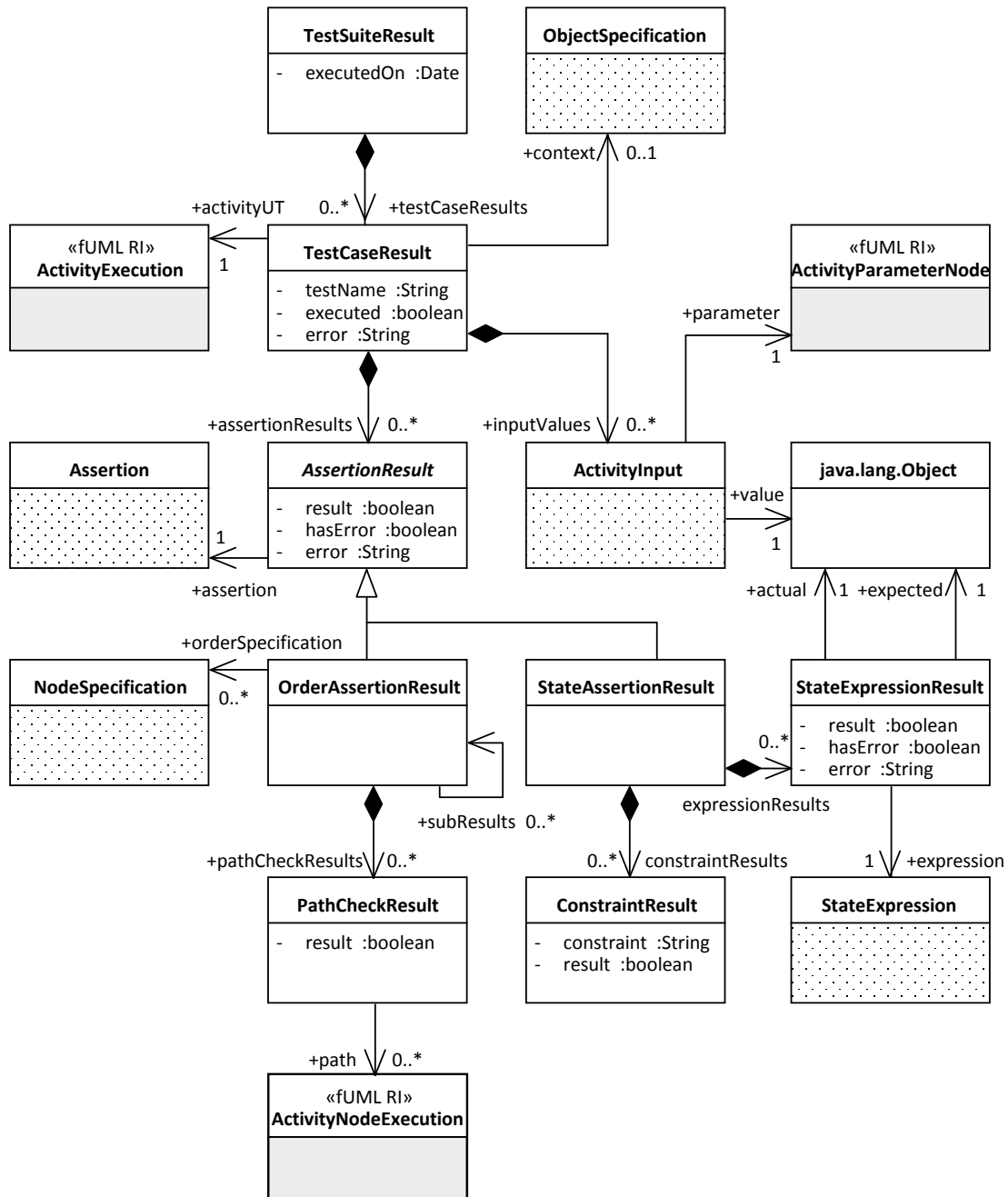


Figure 7.8: Test results model

instance of *TestCaseResult* is created. It contains the name of the test case (attribute *testName*), a reference to the *ActivityExecution* representing the execution of the activity under test in the execution trace (association *activityUT*), and input values set for each input parameter of the activity under test (association *inputValues*). Test case result contains a flag *executed* indicating whether the test case was executed. If an error occurred and the test case was not able to execute, message containing information about the error is recorded (attribute *error*).

For each assertion defined within a test case, an instance of *AssertionResult* is created. Assertion result records the outcome of an assertion evaluation (attribute *result*). If the assertion could not be evaluated, a flag *hasError* is set to true, and the message about the error is recorded (attribute *error*).

If an order assertion has been evaluated, an instance of *OrderAssertionResult* is created. Specification of order which has been evaluated is recorded by the association *orderSpecification*. If any suborder specification has been defined, a list of results for each suborder is recorded (association *subResults*). For each checked path which failed the validation, a result (class *PathCheckResult*) is recorded. This result is used for any counter example of a failing assertion, if found (cf. Section 7.3). The number of failing paths found and added to the assertion result is constrained to a predefined value.

If a state assertion has been evaluated, an instance of *StateAssertionResult* is created. For each state expression defined within the state assertion, a corresponding *StateExpressionResult* instance is created. *StateExpressionResult* contains information about outcome of the evaluation, and the expected and actual values asserted. Furthermore, for each constraint invoked within the state assertion, an instance of *ConstraintResult* is created. *ConstraintResult* contains information about the constraint which was evaluated, as well as the outcome of the evaluation.

## 7.7 ATM Example Revisited

In Listing 7.1 results of running the test case from Listing 6.14 from Chapter 6 are presented.

Listing 7.1: Result of the *Account.MakeWithdrawal* activity test case with exceeding amount

```
1 TestCase: makeWithdrawalFail
2 Activity: Account.MakeWithdrawal
3 Activity context object: accountID
4 Activity input: amount = 150;
5
6 State assertion: always after action successFalse
7   Constraints checked: 2
8   Constraints failed: 1
9     Constraint: NoWithdrawalsCreated
10  State expressions checked: 2
11  State expressions failed: 0
```



Looking at the Listing 7.1 in lines 8-9, we see that the evaluation of the OCL constraint *NoWithdrawalsCreated*, implementing the functional requirement 2c from Chapter 6 for the *Account.MakeWithdrawal* activity, has failed. If we look closely to the activity under test (cf. Figure 5.2), the action *createNewWithdrawal* has no incoming edges, and thus will always be executed, regardless of the input or the initial state of the activity under test.

To fix this defect, it is necessary to introduce an additional control flow from the action *setBalance* to the action *createNewWithdrawal*, ensuring that the new withdrawal instance is only created when the amount provided does not exceed the balance of the account.

In Listing 7.2 results of running the test case from Listing 6.12 from Chapter 6 are presented.

Listing 7.2: Result of the *Account.MakeWithdrawal* activity test case with non-exceeding amount

```

1 TestCase: makeWithdrawalSuccess
2 Activity: Account.MakeWithdrawal
3 Activity context object: accountTD
4 Activity input: amount = 25;
5   Order specification: *, greaterOrEquals, *, setBalance, *, successTrue;
6   Validation result: SUCCESS
7
8   State assertion: always after action successTrue
9   Constraints checked: 1
10  Constraints failed: 0
11  State expressions checked: 2
12  State expressions failed: 1
13  Expression: context.result::balance = 75 / Actual was: -75.0

```

Looking at the Listing 7.2 in lines 12-13, we see that the evaluation of the property state expression checking the balance of the account has failed. Based on the expected and the real value of the balance, it is visible that there is a defect in the activity concerning calculation of the new balance during the withdrawal process. If we look closely to the activity under test (cf. Figure 5.2), the object flows providing input for the action *minus* are wrong, i.e., the object flow pointing to the pin *X* should actually point to the pin *Y* and the object flow pointing to the pin *Y* should actually point to the pin *X*.

To fix this defect, it is necessary to switch the object flows for the action *minus* in the correct way. The corrected version of the *Account.MakeWithdrawal* activity is presented in Figure 7.9.

In Listings 7.3 and 7.4, results of running the test cases from Listings 6.16 and 6.17 are presented, respectively.

## 7. TEST INTERPRETER

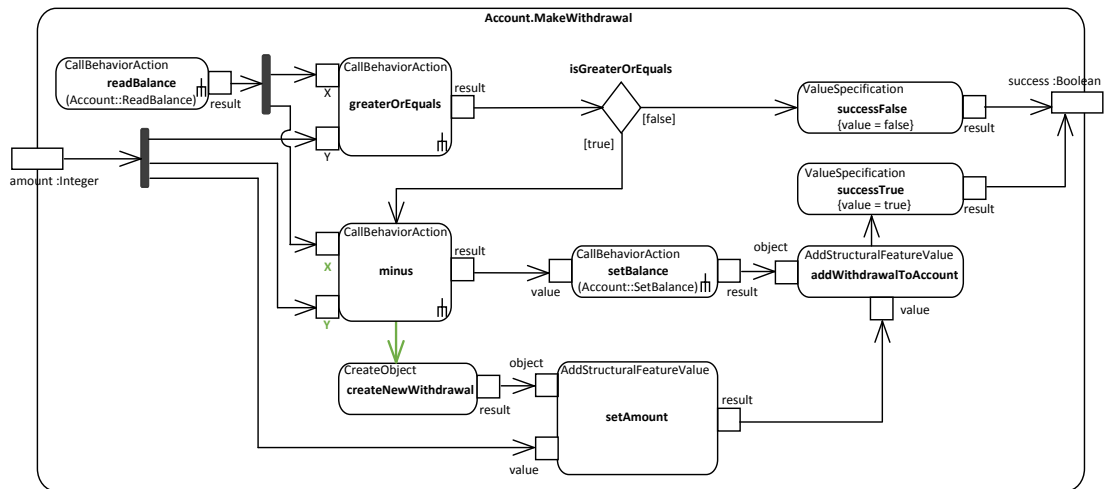


Figure 7.9: The corrected version of the *Account.MakeWithdrawal* activity

Listing 7.3: Result of the *ATM.Withdraw* activity test case with incorrect pin and exceeding amount

```

1 TestCase: atmWithdrawalFail
2 Activity: ATM.Withdraw
3 Activity context object: atmTD
4 Activity input: card = cardTD; pin = 1986; amount = 150;
5
6 Order specification: *, greaterOrEquals, *, successFalse;
7   Validation result: SUCCESS
8
9   State assertion: always after action successFalse
10  Constraints checked: 4
11  Constraints failed: 2
12    Constraint: TransactionEnded
13    Constraint: TransactionRecorded
14  State expressions checked: 2
15  State expressions failed: 0

```

Listing 7.4: Result of the *ATM.Withdraw* activity test case with correct pin and non-exceeding amount

```

1 TestCase: atmWithdrawalSuccess
2 Activity: ATM.Withdraw
3 Activity context object: atmTD
4 Activity input: card = cardTD; pin = 1985; amount = 25;
5
6   State assertion: always after action makeWithdrawal
7   Constraints checked: 4
8   Constraints failed: 2
9     Constraint: TransactionEnded
10    Constraint: TransactionRecorded
11  State expressions checked: 2
12  State expressions failed: 0

```

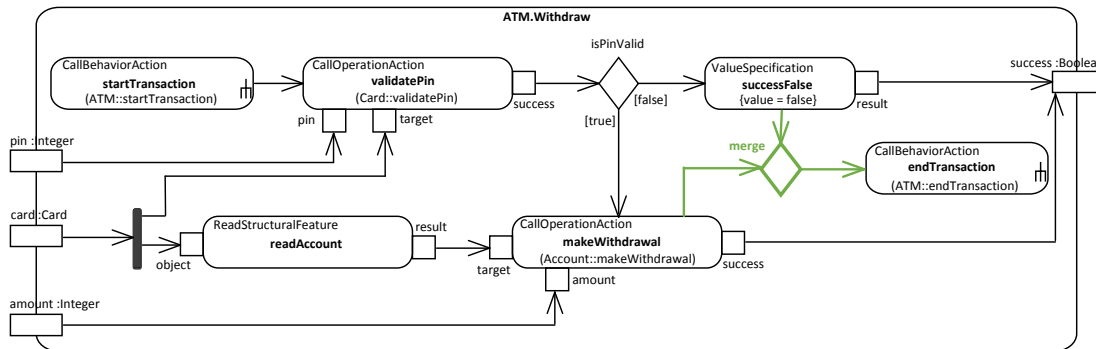


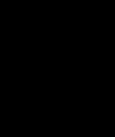
Figure 7.10: The corrected version of the *ATM.Withdraw* activity

As can be seen in Listings 7.3 and 7.4 in lines 8-10, two constraints checking whether the current transaction was removed from the ATM and added to the list of completed transactions of the ATM have failed. As these two operations are performed by the action *endTransaction* in the activity under test, an assumption could be made that something went wrong with execution of this action. If we look closely at the activity diagram in Listing 5.3, there are two incoming control flow edges into the action *endTransaction*.

As described in Chapter 4, an action within an fUML activity can be executed only if it has received control and data tokens on all of its incoming edges. In the *Withdraw* activity from Figure 5.3, the action *endTransaction* can receive a control token from either the action *successFalse* or the action *makeWithdrawal* but not from both, during an execution of the activity, and therefore can never be executed. In order to resolve this fault, it is necessary to introduce a merge node before action *endTransaction*, so that when a control token comes from either *successFalse* or *makeWithdrawal*, it can be executed. The corrected version of the *ATM.Withdraw* activity is presented in Figure 7.10.

For installation instructions of our testing framework in the EMF environment please refer to Appendix A.





# Evaluation

To evaluate the ease of use and usefulness of our testing framework, we have performed a user study with eleven participants. In the following section, we present setup of the user study including a description of the tasks that the participants had to complete, the results of the user study consisting in our observations of the participants during the execution of the tasks, and lessons learned from the user study. Finally, we present a brief comparison of test cases written in our test specification language and using JUnit framework at the code level, accessing the execution trace directly. We have executed the tests and compared the execution times in order to evaluate the overhead introduced by the test specification language interpreter.

## 8.1 User Study

The target group of our testing framework are practitioners in the MDE domain using UML activity diagrams to define the behavior of systems. Thus, in order to obtain relevant results, our selection of participants was based on their background in UML and unit testing. A background in fUML was desirable but not mandatory for participants of the user study.

The user study consisted of four steps: *(i)* an introduction to fUML and our testing framework, *(ii)* a questionnaire regarding the skill level of the participants in using languages relevant for the user study, *(iii)* two tasks that had to be completed with our testing framework, and *(iv)* an opinion questionnaire where the participants rated the ease of use and usefulness of the testing framework for completing the given tasks. The user study was done with each participant separately.

At the beginning of the user study, the participant was given a quick introduction into fUML and our testing framework. This introduction included the most important concepts of fUML comprising fUML's class concepts, activity concepts, and action

Language	no experience	beginner	average	expert
UML Class Diagrams		5	6	
UML Activity Diagrams		8	3	
UML Action Language	3	3	3	2
OCL	1	5	2	3
Unit Testing (e.g., JUnit)	1	3	6	1

Table 8.1: Results of the skills questionnaire

language. Furthermore, the introduction contained a simple exemplary fUML model, which was used to introduce the main concepts of our test specification language.

After the introduction, the participant was given a questionnaire for assessing his/her knowledge of UML, OCL, and unit testing. Among the participants we had post doctoral, doctoral, and master students with different levels of knowledge of these languages. As can be seen in Table 8.1, most of the participants had a good background in UML being slightly more experienced with class diagrams than with activity diagrams. The knowledge of the UML action language was balanced from having no experience to being an expert. Most of the participants declared their experience with OCL at the beginner level, while unit testing knowledge was declared as average by most of the participants.

After completing the skill questionnaire, the participant was asked to complete two tasks with our testing framework. The aim of the first task was to slowly introduce the participant into our test specification language as well as to evaluate its ease of use. In the first task, the participant had to define a test suite implementing given requirements for two given and correct UML activity diagrams. To perform this task, the participant used the testing framework, including the editor for the test specification language and the test interpreter for executing the test cases and providing the test results as console output.

As first task, the participant needed to specify a test scenario with an object to be provided as input to the activity under test, and two test cases with two different order assertions and two different state assertions. The activity was composed of several simple fUML actions, such as *Value Specification Action* and *Read Structural Value Action*.

For the second activity, the participant needed to specify a test scenario with several objects and links, a context object for the activity under test, two state assertions asserting the state at the beginning and at the end of the activity, and one OCL expression for checking the state of the object provided as output of the activity. The activity was composed of several simple fUML actions, and one *Expansion Region* for iterating over a set of objects.

As the second task, the participant was given a defective activity diagram, two test cases testing the activity diagram, and the test results of these test cases. Based on the test cases and test results, the participant had to locate the defects and suggest corrections. With the second task, we aimed at evaluating the usefulness of test cases and test results

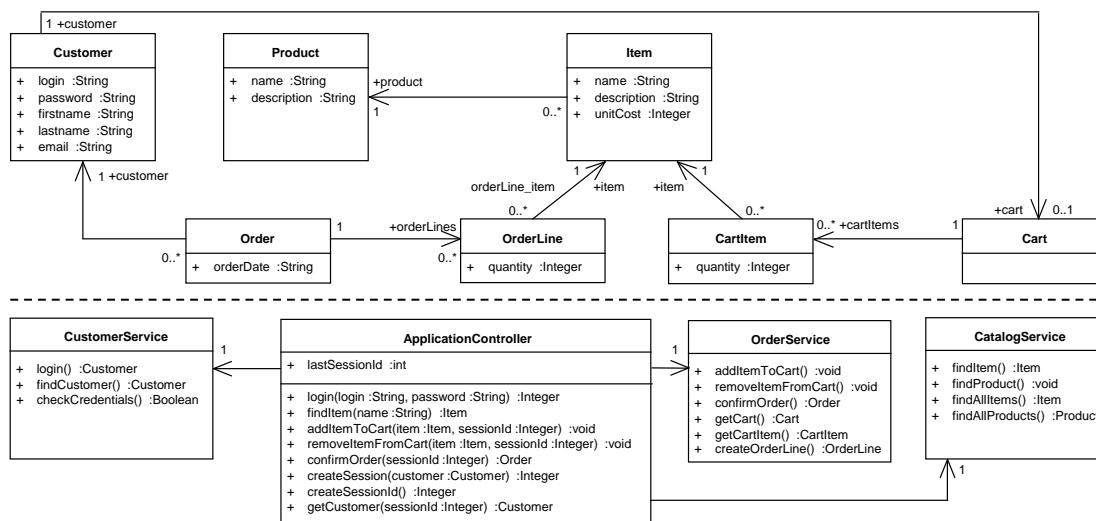


Figure 8.1: Structure and logic layer of the online Web Shop application

with regard to detecting and locating defects in UML activity diagrams. The activity, as in the first task, was composed of several simple fUML actions.

Finally, at the end of the user study, an opinion questionnaire was given to the participants, where each participant rated his/her subjective opinion on the ease of use and usefulness of our testing framework.

### 8.1.1 Provided Material

For performing the user study, we have used an example of an online Web Shop application, whose data and logic layer are presented in Figure 8.1. The Web Shop manages several kinds of entities. A user (class *Customer*) can log in to the shop, browse products (class *Product*), and add items (classes *Item* and *CartItem*) to his/her cart (class *Cart*). Once the user completes selection of the products, by adding items into the cart, he/she can create an order (class *Order*), composed of order lines (class *OrderLine*) created for each item in the cart.

The main component of the service layer is the *ApplicationController*. It is used to create a session for a given user, find items, add items into a cart, remove items from a cart, and confirm orders. For implementing this functionality, the *ApplicationController* uses services provided by the *CustomerService*, the *CatalogService*, and the *OrderService*. Activities implementing the behavior of each relevant operation of the service layer were given in each task.

**Task 1. Writing the Test Cases:** In Figure 8.2 the activity *CheckCredentials* specifying the process of checking credentials of an existing user in the system is presented. This activity is responsible for checking the credentials of a customer when the customer logs

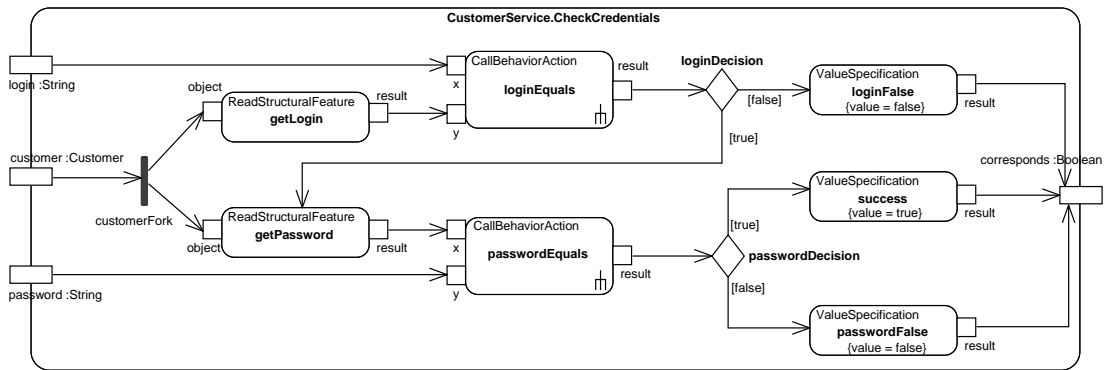


Figure 8.2: Activity *CheckCredentials* of the *CustomerService* class

into the Web Shop. For a given customer, it is checked whether the provided login and password correspond to the login and password associated with the customer. If this is the case, true is returned. In case a wrong login or password was provided, a false value is provided as output of the activity.

The participants were asked to specify a test suite for the activity *CheckCredentials*, that will:

1. Check if given that the correct login and password of an existing customer are provided as input to the activity
  - the actions *loginEquals* and *passwordEquals* are executed.
  - a true value is provided as output of the activity.
  
2. Check if given that the incorrect password is provided as input to the activity
  - the actions *passwordEquals* and *passwordFalse* are executed.
  - a false value is provided as output of the activity.

An example of a correct implementation of a test suite specifying these requirements is presented in Listing 8.1.



Listing 8.1: An example test suite implementing requirements for the *CheckCredentials* activity

```

1 scenario TestData[
2   object customerTD: Customer{
3     login = 'user';
4     password = 'pass';
5   }
6 ]
7 test correctPass activity CheckCredentials(customer = TestData.customerTD,
8   login = 'user', password = 'pass'){
9   assertOrder *, loginEquals, *, passwordEquals, *;
10  finally { corresponds = true; }
11 }
12 test incorrectPass activity CheckCredentials(customer = TestData.customerTD,
13   login = 'user', password = 'wrong-pass'){
14   assertOrder *, passwordEquals, *, passwordFalse, *;
15   finally { corresponds = false; }
16 }

```

In this listing, the presented test suite is composed of one test scenario and two test case. The test scenario, presented in lines 1-6, contains an object of *Customer* class, with some arbitrary values of the properties *login* and *password*. This object is then passed as input into the *CheckCredentials* activity in the test cases in lines 7-8 and 12-13.

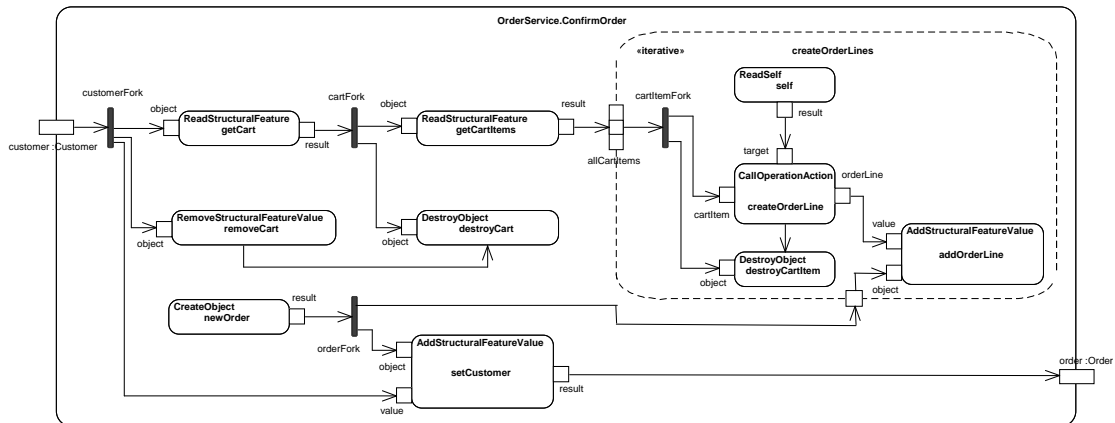
The first test case implements the first part of the requirements, specifying that if the correct login and password are provided as input, the actions *loginEquals* and *passwordEquals* are executed (line 9 in Listing 8.1), and that a true value is provided as output of the activity (line 10 in Listing 8.1).

The second test case implements the second part of the requirements, specifying that if an incorrect password (value *wrong-pass* in line 13 of Listing 8.1) is provided as input, actions *passwordEquals* and *passwordFalse* are executed (line 14 in Listing 8.1), and that a false value is provided as output of the activity (line 15 in Listing 8.1).

In Figure 8.3 the activity *ConfirmOrder* specifying the process of creating an order is presented. The activity retrieves the cart of a given customer, creates a new order, adds an order line into the order for each cart item from the cart, and finally destroys all cart items as well as the cart itself. The order is provided as output of the activity.

The participants were asked to define a scenario consisting of the following objects and links:

- One instance of class *OrderService*.
- Two instances of class *Product* with attribute 'name' set arbitrarily.
- Two instances of class *Item* with different unit costs, associated with the given products.
- Two instances of the class *CartItem* with different quantities, associated with the given items.
- One instance of the class *Cart*, associated with the *CartItem* instances.
- One instance of the class *Customer* associated with the *Cart* instance.

Figure 8.3: Activity *ConfirmOrder* of the *OrderService* class

Once the scenario was specified, rest of the task was to write a test suite that will:

1. Check that the order instance provided as output of the activity is associated with the customer provided as input.
2. Check that the cart has been removed from the customer by the activity.
3. Check that the number of order lines in the newly created order is equal to the number of cart items that were contained by the cart of the provided customer.

Correct implementation of the test scenario is given in Listing 8.2.

Listing 8.2: An example of the test scenario implementing specified requirements for the *ConfirmOrder* activity

```

1 scenario TestData[
2   object orderService: OrderService{}
3
4   object productOne: Product{ name = 'product-A'; }
5   object productTwo: Product{ name = 'product-B'; }
6   object itemOne: Item{ unitCost = 5; }
7   object itemTwo: Item{ unitCost = 10; }
8   link item_product{ source item = itemOne; target product = productOne; }
9   link item_product{ source item = itemTwo; target product = productTwo; }
10
11  object cartItemOne: CartItem{ quantity = 1; }
12  object cartItemTwo: CartItem{ quantity = 5; }
13  link cartItem_item{ source cartItem = cartItemOne; target item = itemOne; }
14  link cartItem_item{ source cartItem = cartItemTwo; target item = itemTwo; }
15
16  object aCart: Cart{}
17  link cart_cartItem{ source cart = aCart; target cartItem = itemOne; }
18  link cart_cartItem{ source cart = aCart; target cartItem = itemTwo; }
19
20  object aCustomer: Customer{}
21  link cart_customer{ source customer = aCustomer; target cart = aCart; }
22 ]

```

An example of the test suite correctly implementing the requirements for the *ConfirmOrder* activity is presented in Listing 8.3.

Listing 8.3: An example of the test suite implementing specified requirements for the *ConfirmOrder* activity

```

1 test confirmOrder activity ConfirmOrder(customer = TestData.aCustomer)
2   on TestData.orderService {
3     initialize TestData;
4     assertState always after action setCustomer {
5       setCustomer.result::customer = TestData.aCustomer;
6     }
7     finally {
8       customer::cart = null;
9       check 'orderLineCheckTwoItems' on order;
10    }
11 }

```

The test suite presented in Listing 8.3 is composed of a single test case, where an object of *Customer* class from test scenario given in Listing 8.2 is provided as input of the activity *ConfirmOrder*. In line 3, an initialize statement is specified which instructs the testing framework to load all defined objects and links from the scenario as the initial state of the system under test. The first requirement of the *ConfirmOrder* activity specifying that the order provided as output of the activity should contain the customer object which was provided as input is specified in lines 4-6 in Listing 8.3.

The second requirement, stating that the cart has been removed from the customer upon activity execution, is specified in line 8 in Listing 8.3. Finally, the last requirement stating that the number of order lines of the newly created order should be equal to the number of the cart items that were contained by the cart, is specified in line 9 in Listing 8.3 as an OCL constraint. The OCL constraint itself, invoked in Listing 8.3 in line 9, is presented in Listing 8.4.

Listing 8.4: OCL constraint for testing the *ConfirmOrder* activity

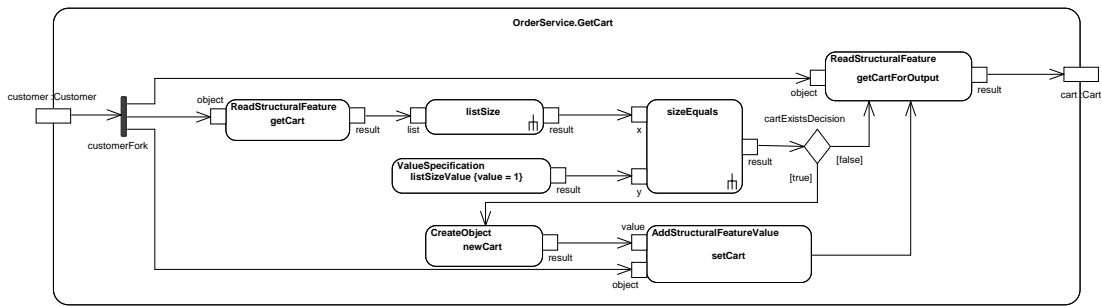
```

1 context Order
2 inv orderLineCheckTwoItems: orderLines -> size() = 2

```

**Task 2. Reading Test Cases and Results:** In Figure 8.4 the activity specifying the process of retrieving a cart of a given customer, is presented. If a given customer is associated with an existing cart, that cart is provided as output of the activity. Otherwise, a new cart object is created, set as cart of the given customer, and provided as output of the activity.

Listings 8.5 and 8.6 show the test cases for evaluating the correctness of the *GetCart* activity. In Listings 8.7 and 8.8 the test results of the given test cases are shown indicating that the activity is not correct. Based on the test cases and test case results, the task is to suggest corrections of the activity, such that the tests will be executed successfully.

Figure 8.4: Activity *GetCart* of the *OrderService* class

Listing 8.5: Test scenario

```

1 scenario OrderServiceScenario[
2   object customer: Customer {login='customer'; password='pass';}
3 ]
4 scenario OrderServiceScenario2[
5   object customer: Customer{login='customer'; password='pass';}
6   object cart: Cart{}
7   link cart_customer{
8     source customer=OrderServiceScenario2.customer;
9     target cart=OrderServiceScenario2.cart;
10  }
11 ]

```

Listing 8.6: Test cases

```

1 test getCartTest activity OrderService.GetCart(customer=OrderServiceScenario.customer){
2   assertOrder *, getCart, *, listSize, *, sizeEquals, cartExistsDecision,
3   newCart, setCart, getCartForOutput;
4   assertState immediately after action getCart {
5     getCart.result = null;
6   }
7   finally {
8     GetCart.cart != null;
9   }
10 }
11 test getCartTest2 activity OrderService.GetCart(customer=OrderServiceScenario2.customer){
12   initialize OrderServiceScenario2;
13   assertOrder *, getCart, *, listSize, *, sizeEquals,
14   cartExistsDecision, getCartForOutput;
15   assertState immediately after action getCart{
16     getCart.result = OrderServiceScenario2.cart;
17   }
18   finally {
19     GetCart.cart = OrderServiceScenario2.cart;
20   }
21 }

```

Listing 8.7: Test case 1 results report

```

1 Test Suite Run: 29-09-2014 15:39:41
2 TestCase: getCartTest
3 Activity: OrderService.GetCart
4 Activity input: OrderService.customer = OrderServiceScenario.customer;
5   Order specification: *, getCart, *, listSize, *, sizeEquals, cartExistsDecision,
6     newCart, setCart, getCartForOutput
7   Number of paths checked: 4
8   Number of invalid paths: 4
9   Failed path: customerFork, listSizeValue, getCart, listSize, sizeEquals, cartExistsDecision
10  Validation result: FAIL
11
12 State assertion: always after action sizeEquals
13 State expressions checked: 1
14 State expressions failed: 1
15   Expression: GetCart.cart != null / Actual was: NULL

```

Listing 8.8: Test case 2 results report

```

1 TestCase: getCartTest2
2 Activity: OrderService.GetCart
3 Activity input: OrderService.customer = OrderServiceScenario.customer;
4   Order specification: *, getCart, *, listSize, *, sizeEquals,
5     cartExistsDecision, getCartForOutput
6   Number of paths checked: 4
7   Number of invalid paths: 4
8   Failed path: customerFork, listSizeValue, getCart, listSize, sizeEquals,
9     cartExistsDecision, newCart, setCart
10  Validation result: FAIL
11
12 State assertion: always after action setCart
13 State expressions checked: 1
14 State expressions failed: 1
15   Expression: GetCart.cart != null / Actual was: NULL

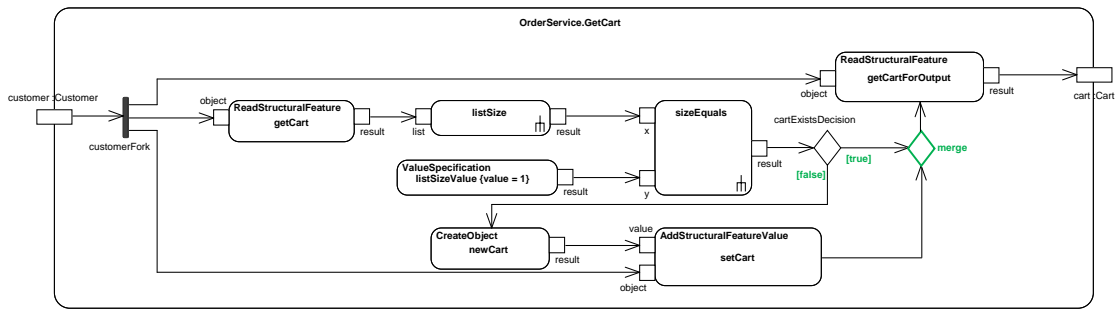
```

In this task, two defects have been introduced. The first defect are incorrectly set guards of the outgoing edges of the decision node, leading to execution of wrong actions depending on whether provided user is associated with a cart or not. The second defect renders the action *getCartForOutput* non-executable, as there is more than one incoming edge to this action. Both defects can be detected from the failing order assertion, and the examples of paths which do not comply with the specified order. It can be seen that the activity executes until the action *getCartForOutput* is reached, but the action itself is not being executed. Furthermore, failing state assertions of the activity output should provide further indication of action *getCartForOutput* not being executed.

In order to correct this defect, a merge node should be introduced before the *getCartForOutput* action. The corrected version of the activity from Figure 8.4 is presented in Figure 8.5.

### 8.1.2 Results

We observed the participants during performing the tasks given in order to find out how quickly they learn to specify test cases for UML activity diagrams using our test

Figure 8.5: Corrected version of the activity *OrderService.GetCart*

specification language, i.e., how easy the test specification language is to use, and whether the test results are useful for detecting and correcting defects in UML activity diagrams. We made the following observations for the first task, where the participants had to define test cases for testing predefined requirements on a given activity diagram.

**Test scenarios.** Most of the participants had problems to understand the purpose of test scenarios, because they tried to define the test scenarios before thinking about and writing the actual test cases. However, after having defined the first test case, the participants understood how to use test scenarios for providing input to the activity under test. For instance, some of the participants tried to specify an object within the test case itself, to be provided as input to the activity.

**Order assertions.** Another frequently observed problem encountered by the participants was to correctly specify order assertions. Several participants specified the expected order of activity nodes incorrectly, as they forgot to use jokers for allowing arbitrary nodes to be executed between two nodes of interest. However, after running the order assertion and reading the failing test result, all participants were able to correct the order assertion. For instance, a participant specified the order assertion from requirement 1 in the first part of the task 1 of the user study as **assertOrder** *loginEquals, passwordEquals*. Furthermore, some participants were not sure whether they should specify the requirement as a one or two assertions, i.e., **assertOrder** *\*,loginEquals, \** and **assertOrder** *\*, passwordEquals, \**, as opposed to **assertOrder** *\*, loginEquals, \*, passwordEquals, \**.

**State assertions.** Another recurring issue was related with specifying state assertions, in particular, with understanding the relation between temporal expressions and state expressions. More precisely, a large number of participants specified for each state expression an own state assertion with its own temporal expression, even though the temporal expressions were semantically and syntactically identical (i.e., only one state assertion would have been sufficient). For instance, the requirement 1 and 2 in the second part of the task 1 could have been specified as a single *finally* state assertion with two state expressions within it implementing both requirements. However, many of the participants were specifying additional finally state assertions for each state expression implementing a concrete requirement.

**OCL expressions.** Several participants had issues with specifying the OCL expression required for one of the test cases. However, this was due to the fact that these participants had little experience with OCL. Connecting the OCL expression with a test case was not an issue for any of the participants.

As can be concluded from these observations, the participants did not experience major difficulties to implement the requirements of the tasks given, and were able to quickly grasp the concepts of the test specification language.

In the second task, the participants needed to read given test cases for a defective UML activity diagram as well as the test results, detect the defects, and suggest corrections of the activity diagram. Thereby, two defects were introduced into the UML activity diagram. One defect consisted in wrong guards for a decision node, which led to the execution of a wrong path. This defect was detectable from the test result of a failing order assertion. The second defect consisted in a missing merge node, which led to an activity node not being executed. This defect was detectable from the test results of a failing order assertion and a failing state assertion. We made the following observations for this task.

**Understanding test cases.** The participants had no problems in understanding the given test cases and their purpose. They were able to describe the requirements tested by the test cases.

**Understanding test results.** Out of the eleven participants, five were able to locate both defects, three were able to locate the first defect only, and three were not able to locate any of the defects. Thereby, we observed that most of the participants had problems in understanding the test result of the defined order assertions. Indeed, several participants reported that it is hard to grasp the meaning of the test result on the first look. However, after having a detailed look on the counter-examples given by the test result (i.e., execution orders of activity nodes invalid with respect to the defined order assertions), most of the participants were able to locate the first defect. For the second defect, knowledge about the necessity of merge nodes for merging alternative execution paths in UML activities was required. Thus, the reason why six participants were not able to detect this defect is their lack of knowledge about the execution semantics of UML activities as defined by the fUML standard.

At the end of the study, the participants had to fill in an opinion questionnaire, where they had to rate how difficult it was for them to accomplish the different tasks. As can be seen from the results of this questionnaire depicted in Table 8.2, our reported observations correspond to the ratings of the participants. According to the participants, it was easy to read and understand the UML class and activity diagrams that were subject of both tasks. Furthermore, it was easy to write test cases for given requirements as part of the first task as well as to read and understand the test cases given in the second task. However, reading and understanding the test results as well as locating and correcting defects of activity diagrams based on these results was more difficult for the participants.

Task	very easy	easy	medium	hard	very hard
Read class diagrams	7	4			
Read activity diagrams	3	7		1	
Write test cases		8		3	
Read test cases	3	4	2	2	
Read test results	3	4	2	2	
Correct activity diagrams	1	3	2	2	3

Table 8.2: Results of the opinion questionnaire

### 8.1.3 Lessons Learned

For completing the first task, the participants needed around one hour in average, while performing the second task took around twenty minutes in average. Furthermore, during the first task, the participants needed less and less time for writing new test cases. These time differences were a result of the fact that with the first task the participants were introduced into the concepts of our test specification language and were learning how to use the different concepts correctly for realizing test cases.

After each written test, the participants were making less mistakes in realizing the next one. By the time they got to the second task, all participants had a clear understanding about all the concepts provided by the test specification language. From this observation, we conclude that our test specification language has a gentle learning curve.

One of the possible improvements that we discovered during the user study is that some concepts of the test specification language, such as the specification of links in test scenarios, could be improved. Furthermore, additional validations by the editor would significantly improve the specification of test cases, as it prevents defects in the test cases themselves.

Although most of the participants were experienced with UML activity diagrams and fUML, many had a lack of knowledge concerning details of the execution semantics of UML activity diagrams. Due to this lack of knowledge, many of the participants had issues with detecting and correcting defects introduced into activity diagrams.

The visualization of test results is crucial for making them understandable and useful for locating defects. Providing more effective means for visualizing test results is part of possible future work. For instance, it would be interesting to investigate the integration of the visualization of test results with UML modeling editors, such that the test results can be presented on the tested activity diagrams themselves. Furthermore, presenting the states of a system caused by the execution of an activity under test in the form of UML object diagrams could be useful to users, as it may provide more insight into the cause of failing test cases.



## 8.2 Comparison with JUnit Tests

In order to compare the complexity of specifying a set of test cases at the model level using our test specification language, as opposed to specifying the same set of test cases at the code level using the JUnit testing framework, as well as to evaluate the performance overhead effecting the execution times induced by our test interpreter, we have implemented a set of test cases using JUnit framework [MH03] at the code level, and measured their execution times. The test cases specified same requirements from the user study, and were specified using the JUnit testing framework by directly invoking assertions on the execution trace.

Regarding the functional requirements, only difference between the test cases in JUnit and corresponding test cases in the test specification language, was that we haven't included the evaluation of order assertions considering parallelism, but rather just a single default execution recorded in the trace by the virtual machine. Considering the parallelism in JUnit test cases would require a lot of additional coding, and to our knowledge would not contribute to the evaluation, as the same mechanism would have to be used in both cases.

The user study was implemented by three classes. A utility class providing means for setting up the scenarios, executing the activity under test, and retrieving objects and links produced by actions and activity had around 450 lines of code. If it was to include evaluation of order assertions considering the parallelism, it would grow substantially, thus the complexity reduction by using the testing framework at the model level is already quite significant.

Additionally, two classes implementing JUnit test cases for the first and second task from the user study were defined. In the first task, as described in previous sections, the functionality for checking the credentials and confirming an order in the system was under test. This task was implemented in JUnit as three test cases, with a bit less than 200 lines of code.

In the second task, as described in previous sections, the functionality for retrieving a cart for a given customer was under test. This task was implemented in JUnit as two test cases, with around 150 lines of code.

On the other hand, the test cases defined in the test specification language were around 80 lines of code long in total, leading to the significant reduction of their complexity. Moreover, the test cases specified at the model level are much more easy to read and understand, as they do not expose the test designer to the intricacies of the inner workings of the execution engine, as well as the trace model.

By reducing the size of the test cases, and abstracting the intricacies of the fUML virtual machine and the execution trace from the test designer, the maintainability of the test cases is improved. As the test cases are at the model level, and the assertions are more concise and directly refer to model elements of the system under test, they can be more easily changed when new requirements arise, or the models under test are changed.

Furthermore, we have executed the defined JUnit and test language test cases, and compared their execution times. The comparison of number of lines of code, as well as the running times between JUnit and testing framework test cases are given in Table 8.3.

Lines of Code		
	JUnit	Testing Framework
Utility classes	450	none
Task 1	200	23
Task 2	150	60

Running Times (ms)		
	JUnit	Testing Framework
Setup	3773	6547
Check Credentials Correct	78	76
Check Credentials Incorrect	16	8
Confirm Order	52	43
Get Cart (no cart)	78	72
Get Cart (with cart)	19	12

Table 8.3: Comparison of JUnit and the Testing Framework test cases

As can be seen from the Table 8.3, running times of test cases in JUnit and the testing framework differ slightly. However, the time required to setup the test cases and models in JUnit and testing framework differ significantly. This setup time in the testing framework is composed of time required for converting the UML model under test into the fUML, converting and loading test scenarios into the VM, and converting the test cases from the test language into an internal representation. In case of JUnit, the setup time consists only of converting UML model under test into the fUML, as the test cases are specified directly in Java programming language.

## Conclusion and Future Work

In this thesis, we presented contributions towards addressing the lack of testing facilities for UML models, based on the precise and standardized specification of the semantics of a subset of UML called fUML [Obj11], and an interpreter capable of executing fUML conformant models. The main contributions of this thesis are an executable test specification language and an environment for testing fUML models. By leveraging the semantics of UML standardized by the OMG in the fUML standard, and building on an extended version of a virtual machine for fUML models (cf. Chapter 4), we have developed a test specification language and a test interpreter, providing means for improving the quality of the developed models, enabling detection and correction of defects at the model level early in the design stage. In the following, we summarize the contributions elaborated in the course of this thesis, as well as conclusions derived from their evaluation.

**Contribution 1: Design of a dedicated test specification language for fUML models.** Our test specification language enables development of test cases for fUML activities at the model level, composed of assertions on the state of the execution of an activity under test, as well as assertions on the order of execution of activity nodes within the activity under test.

An activity might require input in form of objects provided to its parameter nodes, in order to be executed. Furthermore, system might be in a certain initial state when an activity is being invoked during a test. For this purpose, our test specification language enables to define *test scenarios* composed of objects, their attribute values, and links between them, which can be set as initial state of the system under test, as well as provided as input to an activity under test.

User can specify state assertions to check objects, their attribute values, and links between them, that were consumed or produced by a certain activity node or an activity parameter node, at the certain point in time of the activity execution. A state assertion is composed

of a time frame, specifying the observed part of execution of the activity under test, and a set of expressions which are checked within the specified time frame. Specification of time frames within the state assertions can be realized by use of a set of temporal operators and quantifiers with the combination of activity nodes representing a beginning and end of an observed time frame (cf. Chapter 6).

Beside expressions for directly checking the state of objects, user can specify OCL constraints which should be evaluated on the state of the system under test at some point in time of the activity execution. OCL enables specification of complex expressions involving operations such as iteration and calculation over objects and their attribute values. Furthermore, OCL constraints can be used for specifying time frames determining the part of execution for which state assertions are checked.

Assertions on the order of execution of activity nodes can be specified for both absolute as well as relative paths of an activity execution. An order assertion is composed of a list of nodes, specified in order in which they are expected to be executed. For specifying a relative order of execution, special escape characters can be used, namely `_` and `*`, for skipping one or more nodes in an execution path.

**Contribution 2: Development environment enabling to create test cases more efficiently using a dedicated test editor.** We have developed an editor for more efficiently specifying the test cases on a UML activity under test. The editor was developed using Eclipse Modeling Framework and Xtext, for creating text based domain specific languages. The editor supports direct linking of test language concepts to the elements of the UML model under test, auto-completion, scoping, and validation (cf. Chapter 6).

**Contribution 3: Framework for execution and evaluation of the test cases.** We have developed a test interpreter capable of executing the test cases created using our test specification language (cf. Chapter 7). The test interpreter is built on top of an extended version of the fUML virtual machine, which records the trace of execution of an activity under test. Once the activity under test is executed, the specified assertions from the test cases are evaluated by analyzing the trace of execution of the activity under test. Thereof, test results indicating which assertions succeeded and which failed are produced and presented to the user.

Test case evaluation process executed by the test interpreter is composed of several steps. At the beginning, specified test scenario is loaded into the virtual machine, and the activity under test is executed with the specified input. Thereof, a trace containing information regarding which objects were produced or modified by the activity execution, as well as the chronological and logical order of nodes within the activity under test is recorded.

Next, each assertion in the test case is evaluated by analyzing the trace of execution. State assertions are evaluated by selecting and evaluating the objects and their attribute values from the trace against the specified values from the assertions. Furthermore, information regarding the chronological and logical order of execution of activity nodes

---

is analyzed to evaluate specified order assertions. Validation of order assertions takes into account possibility of specifying concurrent paths of execution within an activity. Concurrency within an activity can be modeled using *fork/join* constructs, or several *starting nodes* (nodes without incoming edges) or multiple outgoing control flows of actions.

In order to support complex expressions on the state of execution of an activity under test, our testing framework supports the specification and evaluation of OCL expressions within defined test cases. For defining and evaluating OCL expressions within test cases, we made use of the DresdenOCL framework (cf. Chapter 7). DresdenOCL supports integration of specifying and evaluating OCL constraints on any modeling language developed using EMF technologies.

**Contribution 4: Test Results Model.** Once a test case is evaluated by the testing framework, test results are produced and presented to the user (cf. Chapter 7). The main component of the test results model is a test suite result composed of test case results for each evaluated test case of an executed test suite. Test case result contains information such as the name of the activity under test, input provided to the activity, and results of each evaluated assertion within the test case (cf. Chapter 7).

We have performed a user study with eleven participants, in order to evaluate the *ease of use* and *usefulness* of our test specification language and the testing framework (cf. Chapter 8). During the study, most of the participants found specifying the test scenarios and the test cases on an UML model under test intuitive and easy to use. There were problems with understanding the test results and applying necessary actions to correct any defects during the testing process, however this was in most cases due to the level of familiarity of a participant with the fUML standard. Based on the user study, we were able to do some improvements to the test specification language and the test interpreter. The study has shown the applicability of the approach.

In the rest of this section we will discuss some limitations of the testing framework, as well as some possible directions of the future work. These limitations and future work directions are based on our experience during building the testing framework, as well as the results of the performed user study (cf. Chapter 8).

**Effect of concurrency on object flows.** Current version of the testing framework considers concurrency in an activity under test during an order assertion evaluation. However, the state assertions are evaluated only against the single activity execution, as produced by the virtual machine. In order to improve the scope of testing capabilities of the testing framework, as well as to improve the test results produced, it is necessary to consider several execution paths when evaluating the state assertions.

**Support for deep call hierarchies** Beside missing to take into account object flows when evaluating the effect of concurrency on an activity execution, current implementation of the testing framework doesn't support activity call hierarchies deeper than a single level. In other words, it is only possible to specify order assertions on activities invoked by

the activity under test. However, it might be useful to be able to specify order assertions on deeper levels.

As there can be potentially huge number of execution paths of an activity under test, for a given input, it would be impossible or at least inefficient to perform validation of state assertions on each execution path. It would be useful to investigate how concurrency might effect actions within an activity under test which produce or modify the state of the system considered by a specified state assertion. Once such actions are identified, execution paths of the activity under test on which re-ordering of such actions could take place should be validated against specified state assertions.

**Mocking of called activities.** Mocking is a very well known software programming technique, used in unit testing for isolating a unit under test (e.g., a method or an object), from any existing dependencies whose execution might influence the test results. In software programming, a mock is usually an object used in place of a real one, and whose behavior can be manipulated for the purpose of a unit test. To achieve a similar goal when testing an activity, it might be necessary to isolate it from any existing called activity. In the current version of the testing framework, this can be accomplished only by modifying the model of the activity under test. Therefore, it would be interesting to investigate possibility of specifying activity mocks within a test case, which could then be used to replace the dependencies of the activity under test during the test execution.

**Integration of OCL syntax with the test specification language.** As described in Chapter 6, the test specification language supports evaluation of OCL constraints on set of execution states, as well as their use for specification of the time frame of a state assertion. However, in the current version of the test specification language, the OCL constraints have to be specified outside of a test case, within an OCL file, and then invoked by the name of the constraint from within a state assertion. This is not only an inconvenience, but it might lead to test case execution failure, as there is no validation of the relation between the specified state assertion and the invoked OCL constraint. Therefore, it would be interesting to investigate possibility of integrating the OCL constraints directly within the test specification language.

**Test coverage analysis.** Test coverage represents a measure of the proportion of a program exercised by a test suite, usually expressed as a percentage. This typically involves collecting information regarding which parts of a program are actually executed during a test suite run. There are several different test coverage criteria, such as statement coverage, branch coverage, and path coverage, and a considerable research work done in this area [DeM89, HJK<sup>+</sup>11, PAOC13]. It would be an interesting line of research to investigate how test coverage can be defined and analyzed for fUML models, for a defined test suite at the model level.

**Corrective feedback.** Another interesting line of research would be to investigate possible application of model slicing techniques for providing a corrective feedback to the user when certain assertion of a test case for an fUML model fails. Calculating a subset of a model under test, affecting the elements of the model asserted by a test case,

---

represents a so called model slicing technique [HBD03, ACH<sup>+</sup>13, SCWM10, RWMR13]. For instance, if a state assertion for a certain activity node fails during a test suite run, it might be possible to isolate a subset of activity nodes which directly influence the variable from the state assertion and provide more information what might have caused the failure. If and how this could be done represents one possible research direction.

**Test case generation.** Another possible future research direction is to apply model based testing approaches to generate test cases for fUML activities based on defined coverage criteria. An overview of model based testing approaches, for automating the process of generation and execution of test suites at the code level, was given in Chapter 3.

**Integration of test results for better visualization.** Integration of test results of running a test suite with JUnit for instance, or with a model editor, might provide better visualization of test results and therewith improve the understandability and usefulness of test results. Furthermore, visualization of states created during the model execution might provide more insight into what led to certain assertion failure. This also presents one possible direction of further research.





# Installing Eclipse Environment and Running the Testing Framework

## A.1 Installing the Environment

The following tools and plugins are required and have been tested for building and running the Testing Framework:

- Java JDK version: 1.7
- Eclipse Modeling Framework version: Kepler SR2
- Xtext Framework with Xbase version: 2.5
- EMFText (prerequisite for DresdenOCL) version: 1.4.1
- DresdenOCL version: 3.3.0

Once the Java JDK and Eclipse have been downloaded and installed, it is necessary to install the Xtext framework. This can be done through Help->Install Modeling Components, and then by selecting Xtext and clicking Finish.

In order to install DresdenOCL plugin successfully, first the EMFText plugin needs to be installed. This can be done through Help->Install New Software dialogue. In the dialogue box, click Add, and add the appropriate information (name: EMFText, location: <http://emftext.org/update>), and click Finish.

Finally, installing the DresdenOCL plugin can be done through Help->Install New Software dialog. In the dialogue box, click Add, and add the appropriate information (name: DresdenOCL, location <http://www.dresden-ocl.org/update/kepler/>), and click Finish.

At this point, the environment should be ready for the Testing Framework. The framework is available under the open source public license, and can be checked out from GitHub repository at <https://github.com/moliz>. Once the project is checked out from the repository, it can be built and run as an Eclipse application from within Eclipse IDE.

### A.2 Setting up a Project

Create standard Java project by selecting New->Java Project from package explorer. Once the project is created, import the UML model under test into a package which is on the build path within the project (e.g., src). Also, if you want to use an OCL file where the OCL constraints for the test case will be defined, the file should be created or imported into a package which is on the build path within the project (e.g., src).

If you intend to use the OCL constraints, the OCL file with the constraints has to be loaded into the DresdenOCL tool. This is done by selecting the UML file in the package explorer, and selecting DresdenOCL->Load as model... and selecting 'UML Class Diagram' as the metamodel of the loaded model, and clicking Finish.

Once the model and OCL file are imported, create a new file by selecting New->'File' inside a package which is on the build path within the project (e.g., src). Make sure to give the file extension '.umltest'. The Eclipse will ask you to add the Xtext nature to the project - select yes.

### A.3 Running the Test Cases

Once the test suite has been defined, it can be run by selecting the Run->Run Configurations... Double-click the fUML TestSuite in the left pane, and select the UML model file, test suite resource file, and the OCL resource file. Note that the OCL file is optional. Furthermore, to obtain a counter example in case of a failing order assertion, OrderAssertion Brute Force should be checked.

In the top part of Figure A.1 an example of a test case written in our test editor is presented. In the middle, a run dialog is presented, where the user can specify the location of the UML model file holding the model under test, the test suite resource file holding the test suite to be run, and an OCL resource file where any OCL constraint that is invoked within the test suite is defined. Finally, in the bottom part of the figure, results of running the test suite printed out in a console view are presented.

Figure A.1: Example of running a test suite

The screenshot shows an IDE with the following components:

- Package Explorer:** Shows a project structure with packages like `org.modelexecution.funitesting.examplesu`, `src`, `tests`, `customerService.umitest`, `orderservice_confirmorder.umitest`, `orderservice_getcat.umitest`, `petstore.di`, `petstore.mutation`, `petstore.oed`, and `petstore.uml`.
- Editor:** Contains the following Java code:
 

```

import petstore.data.*;
import petstore.logic.*;

scenario TestData {
  object customerOne: Customer(login="customerOne"; customer.password="pass1");
  object customerTwo: Customer(login="customerTwo"; customer.password="pass2");
  object customerService: CustomerService{}
}

//tests for CheckCredentials activity
test checkCredentialsSuccess activity CustomerService.CheckCredentials(customer=TestData.customerOne,
customerService.CheckCredentials(login="customerOne", CustomerService.CheckCredentials.password="pass1")) {
  initialize TestData;
  assertOrder *, CustomerService.CheckCredentials.loginEquals *, CustomerService.CheckCredentials.passwordEquals *,
  finally{
    CustomerService.CheckCredentials.corresponds = true;
  }
}

test checkCredentialsPassFail activity CustomerService.CheckCredentials(customer=TestData.customerOne,
CustomerService.CheckCredentials(login="customerOne", CustomerService.CheckCredentials.password="pass2")) {
  assertOrder *, CustomerService.CheckCredentials.passwordEquals, ..., CustomerService.CheckCredentials.passwordFalse;
  CustomerService.CheckCredentials.corresponds = false;
}
      
```
- Console:** Shows the execution output:
 

```

Test Suite Run: 02-03-2015 18:22:49
Testcase: checkCredentialsSuccess
Activity: CustomerService.CheckCredentials
Activity Input: customer = customerOne; login = customerOne; password = pass1;
Order specification: *, loginEquals, *, passwordEquals, *
Validation result: SUCCESS
State assertions: ALWAYS AFTER action success
State expressions checked: 1
State expressions failed: 0
Testcase: checkCredentialsPassFail
Activity: CustomerService.CheckCredentials
Activity Input: customer = customerOne; login = customerOne; password = pass2;
Order specification: *, passwordEquals, ..., passwordFalse
Validation result: SUCCESS
State assertions: ALWAYS AFTER action passwordFalse
State expressions checked: 1
      
```
- Run Configurations Dialog:** Titled "Run Configurations" and "Create, manage, and run configurations". It shows:
  - Name: customerServiceTests
  - Environment: Common
  - UML Model Resource: /org.modelexecution.funitesting.examplesu.userstudy.c
  - Test Suite Resource: /org.modelexecution.funitesting.examplesu.userstudy.c
  - OCL Resource: /org.modelexecution.funitesting.examplesu.userstudy.c
  - Order Assertion Breakforce:
  - Filter: Filter text
  - Selected items: UML TestSuite, appControllerTests, confirmOrderTests, CustomerServiceTests, getCatTests, Java Applet, Java Application.
  - Filter matched 21 of 21 items



# Xtext Implementation of the Test Specification Language

In this subsection we will present each part of the grammar of our test specification language implemented in Xtext. In our implementation, we have specified the grammar beforehand, and therefore the Ecore metamodel of the language was generated from the grammar.

In Listing B.1 main component for declaring the qualified name of the language and imported metamodels is presented. As can be seen on line 1 in Listing B.1 Xbase<sup>1</sup> is introduced into the grammar. Xbase represents a reusable expression language containing mechanisms for performing type checking according to the Java type system, as well as many default implementations of UI aspects. Furthermore, Xbase expression language contains many features such as object instantiation, method invocation, exceptions and many more.

Listing B.1: Test language grammar declaration

```
1 grammar org.modelexecution.fumltesting.uml.UmlTestLang with org.eclipse.xtext.xbase.Xbase
2 import "http://www.eclipse.org/uml2/4.0.0/UML" as uml
3 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4 generate umlTestLang "http://www.modelexecution.org/fumltesting/uml/UmlTestLang"
```

The two import statements on lines 2-3 in Listing B.1 are used for importing elements of UML and Ecore metamodels into the grammar, which enable referencing UML model elements from elements of the test specification language. Finally, *generate* statement instructs to Xtext to generate the metamodel of the language based on the grammar definition.

---

<sup>1</sup><http://wiki.eclipse.org/Xbase>

The main components of the language are declared in Listing B.2. The starting point in the language is a *test suite*. A test suite is composed of import statements, scenarios and test cases.

Listing B.2: Test suite grammar

```
1 UMLTestSuite:  
2 (imports += Import)* (scenarios += UMLScenario)* (tests += UMLTestCase)*;
```

Import statement grammar rule is presented in Listing B.3.

Listing B.3: Import statement grammar

```
1 Import:  
2 'import' importedNamespace = QualifiedNameWithWildcard;
```

Import grammar is composed of the keyword *import* and the qualified name of the imported UML package where the model under test is located. The rule *QualifiedNameWithWildcard* is defined by the Xbase library.

A test scenario is used for specifying objects and links that can be used to provide input to the activities under test, or as expected value of assertions within a test case. The grammar of a test scenario is presented in Listing B.4.

Listing B.4: Test scenario grammar

```
1 UMLScenario:  
2 'scenario' name = ID '[' (objects += UMLObjectSpecification)* (links += UMLLink)* '];
```

A test scenario is composed of a name, a number of objects, and a number of links. Grammar for specifying an object is presented in Listing B.5.

Listing B.5: Object specification grammar

```
1 UMLObjectSpecification:  
2 'object' name = ID ':' type = [uml::Class|QualifiedName] '{' (attributes += UMLAttribute)* '}';
```

Object specification grammar is composed of a name of the object, a UML class declared as type of the object, and a number of attribute declarations. Grammar for specifying an attribute value of an object is presented in Listing B.6.

Listing B.6: Attribute specification grammar

```
1 UMLAttribute:  
2 att = [uml::Property|QualifiedName] '=' value = UMLValue ';;
```

Attribute value grammar rule is composed of a UML property which is a type of the attribute, = assignment operator, and a value assigned to the attribute. In Listing B.7 grammar for specifying simple and object values is presented.

---

### Listing B.7: Simple and object value grammar

```
1 UMLValue:
2   UMLSimpleValue | UMLObjectValue;
3 UMLSimpleValue:
4   (negative ?= '-' )? value = XLiteral;
5 UMLObjectValue:
6   value=[UMLObjectSpecification|QualifiedName];
```

As can be seen in Listing B.7 a value can be either a simple value or an object value. A simple value is a literal (e.g., string, integer or boolean) defined in Xbase, with an optional *minus* sign for specifying negative integers. Object value is simply a reference to an object defined within the test specification language.

Finally, the grammar for a link specification which can be defined within a test scenario between two objects, is presented in Listing B.8.

### Listing B.8: Link specification grammar

```
1 UMLLink:
2   'link' assoc=[uml::Association|QualifiedName] '{'
3     'source' sourceProperty=[uml::Property|QualifiedName]
4       '=' sourceValue=[UMLObjectSpecification|QualifiedName] ';'
5     'target' targetProperty=[uml::Property|QualifiedName]
6       '=' targetValue=[UMLObjectSpecification|QualifiedName] ';'
7   '}';
```

Link grammar is composed of the association of which the link is an instance, *source* keyword followed by the reference to the source property and source object representing the source end of the link, and the *target* keyword followed by the target property and the target object representing the target end of the link.

Next component of a test suite is a test case. A test case is composed of a name, a reference to an activity under test, activity input elements for each input required by the activity under test, an optional context object reference, an initialize statement and a number of assertions. As can be seen from Listing B.9 an initialize statement is composed of the *initialize* keyword, and one or more references to the test scenarios separated by comma.

### Listing B.9: Test case grammar

```
1 UMLTestCase:
2   'test' name = ID 'activity' activityUnderTest = [uml::Activity|QualifiedName]
3     ('(' inputs += UMLActivityInput (',' inputs += UMLActivityInput)* ')')?
4     ('on' contextObject = [UMLObjectSpecification|QualifiedName])? '{'
5     ('initialize' (initScenarios += [UMLScenario]) (',' (initScenarios += [UMLScenario]))* ';')?
6     (assertions += UMLAssertion)*
7   '}';
```

Activity input grammar is presented in Listing B.10. Activity input is composed of an activity parameter node, assignment operator =, and a value provided as input to the parameter node.

Listing B.10: Activity input grammar

```

1 UMLActivityInput:
2   parameter = [uml::ActivityParameterNode|QualifiedName] '=' value = UMLValue;

```

As described in Chapter 6, there are three types of assertions that can be specified in the test specification language. Grammar specifying existing kinds of assertions is presented in Listing B.11.

Listing B.11: Assertions grammar of the test specification language

```

1 UMLAssertion:
2   UMLOrderAssertion | UMLStateAssertion | UMLFinallyStateAssertion;

```

The grammar for the order assertion is presented in Listing B.12.

Listing B.12: Order assertion grammar

```

1 UMLOrderAssertion:
2   'assertOrder' order = UMLNodeOrder ';' ;
3 UMLNodeOrder:
4   nodes += UMLNodeSpecification (',' nodes += UMLNodeSpecification)*;
5 UMLNodeSpecification:
6   node = [uml::ActivityNode|QualifiedName] (':' size = XNumberLiteral)?
7   ('(' subOrder = UMLNodeOrder ')')? | joker='*' | joker='_';

```

An order assertion is composed of the keyword *assertOrder* and one or more node specifications separated by comma. A node specification is either an activity node owned by the activity under test, or a joker (star or underscore). If an activity node is specified as element of an order assertion, then an optional size argument can be specified signifying the occurrence of the node in the path (can be used for loops). Furthermore, for each activity node it is possible to specify an optional suborder, in case an activity is invoked from the activity under test.

Grammar of a state assertion is presented in Listing B.13. A state assertion is composed of the keyword *assertState*, a temporal quantifier, a temporal operator, a reference point for specifying one end of the time frame, an optional *until* point for specifying an additional end of the time frame, and a number of expressions for checking the selected states.

Listing B.13: State assertion grammar

```

1 UMLStateAssertion:
2   'assertState' quantifier = UMLTemporalQuantifier operator = UMLTemporalOperator
3   referencePoint = UMLReferencePoint ('until' untilPoint = UMLReferencePoint)?
4   '{' (checks += UMLCheck)* '}';

```

Grammar for specifying a temporal quantifier and a temporal operator is presented in Listing B.14. For specifying the grammar of quantifier and operator we make use of Xtext enumerations [xte13].



---

#### Listing B.14: Temporal operator and temporal quantifier grammar

```
1 enum UMLTemporalOperator:  
2   after | until;  
3 enum UMLTemporalQuantifier:  
4   always | sometimes | eventually | immediately;
```

For specifying a reference point of a time frame it is possible to refer to an action within the activity under test or to invoke evaluation of an OCL constraint, as described in Chapter 6. In Listing B.15 grammar for specifying reference points is presented.

#### Listing B.15: Reference point grammar

```
1 UMLReferencePoint:  
2   UMLActionReferencePoint | UMLConstraintReferencePoint;  
3 UMLActionReferencePoint:  
4   'action' action = [uml::Action|QualifiedName];  
5 UMLConstraintReferencePoint:  
6   'constraint' constraintName = XStringLiteral;
```

Within a state assertion a number of expressions for checking the state can be specified. There are two kinds of expressions that can be specified within a state assertion, as presented in Listing B.16.

#### Listing B.16: Grammar specifying possible kinds of state expressions for checking the state in a state assertion

```
1 UMLCheck:  
2   UMLConstraintCheck | UMLStateExpression;
```

The constraint check is used for specifying invocation of an OCL constraint of the state selected by a state assertion. Grammar for specifying an invocation of an OCL constraint is presented in Listing B.17.

#### Listing B.17: Grammar for specifying an invocation of an OCL constraint

```
1 UMLConstraintCheck:  
2   'check' constraintNames += XStringLiteral (',' constraintNames += XStringLiteral)*  
3   ('on' object = [uml::ObjectNode|QualifiedName])? ';' ;
```

An invocation of an OCL constraint is composed of the keyword *check*, a number of constraint names separated by comma, and an optional *on* declaration for limiting the evaluation of the constraints on a single object within the asserted states.

Beside the constraint check, within a state assertion it is possible to specify number of state expressions which are specialized into either *object state expression* or *property state expression*, as presented in Listing B.18.

Listing B.18: Grammar for specifying state expressions

```

1 UMLStateExpression:
2   UMLObjectStateExpression | UMLPropertyStateExpression;
3 UMLObjectStateExpression:
4   pin = [uml::ObjectNode|QualifiedName] operator = UMLArithmeticOperator value = UMLValue ';' ;
5 UMLPropertyStateExpression:
6   pin = [uml::ObjectNode|QualifiedName] '::' property = [uml::Property|QualifiedName]
7   operator = UMLArithmeticOperator value = UMLValue ';' ;

```

Object state expression is composed of a an input or output pin of the activity under test or an action within the activity under test, the arithmetic operator, and a value compared to the object provided as input or output of the specified pin. Property state expression, compared to the object state expression, contains an additional operator ('::') and a property of the object being checked.

The grammar for specifying arithmetic operator is presented in Listing B.19.

Listing B.19: Arithmetic operator grammar

```

1 enum UMLArithmeticOperator:
2   equal = '=' | not_equal = '!=' | greater = '>' | smaller = '<' | greater_equal = '>=' |
3   smaller_equal = '<=' | includes = 'includes' | excludes = 'excludes';

```

Similarly as for the temporal operators and quantifiers, we make use of Xtext enumerations for specifying the grammar of the arithmetic operator.

The last kind of assertion possible to specify in a test case is the *finally state assertion*. In Listing B.20 the grammar for specifying the finally state assertion is presented.

Listing B.20: Finally assertion grammar

```

1 UMLFinallyStateAssertion:
2   'finally' '{' {UMLFinallyStateAssertion} (checks += UMLCheck)* '}' ;

```

Finally assertion is composed of the keyword *finally* and a number of expressions for checking the selected states. In this rule an action for making an object of the finally state assertion is specified (i.e., *{UMLFinallyStateAssertion}*), which is necessary to ensure that a finally state assertion with an empty body would cause the object creation in the AST.

# List of Algorithms

1	Generation of execution trees for each starting node from a given activity execution $e$ . . . . .	87
2	Generation of an execution tree for a given activity node execution $n$ . . .	91
3	Retrieve potential successors of a given activity node execution $n$ . . . . .	91
4	Check if a potential successor of a given activity node execution $m$ can be added as a child of a given activity node execution $n$ . . . . .	92

# List of Figures

2.1	Metamodeling levels of the MDA standard, modified version of a figure taken from [Obj15] . . . . .	15
2.2	Model transformations in the MDA standard, modified version of a figure taken from [ATL06] . . . . .	16
3.1	Summary of presented UML testing approaches . . . . .	30
4.1	Excerpt of the UML metamodel containing the basic concepts of activity diagrams [Obj15] . . . . .	45
4.2	Modeling concepts of UML activity diagrams [May11] . . . . .	46
4.3	Overview of fUML execution environment extensions [May14] . . . . .	50
5.1	Structure of the ATM system . . . . .	53
5.2	Activity <i>MakeWithdrawal</i> of the Account class . . . . .	54
5.3	Activity <i>Withdraw</i> of the ATM class . . . . .	56
5.4	Overview of the testing framework . . . . .	60
6.1	Usage of temporal operators and quantifiers in state assertions . . . . .	68

7.1	Excerpt of the execution trace model [May14]	85
7.2	An example activity with several starting and several ending nodes	87
7.3	An excerpt of an execution trace produced by the fUML virtual machine for the activity from Figure 7.2	88
7.4	Execution tree generated from the execution trace from Figure 7.3, with the node <i>initial</i> as its root	89
7.5	Adjacency matrix of the activity from Figure 7.2, generated from the execution trace from Figure 7.3 produced by fUML virtual machine	93
7.6	Excerpt of sequence execution trace model	95
7.7	Excerpt of the package architecture of DresdenOCL	98
7.8	Test results model	101
7.9	The corrected version of the <i>Account.MakeWithdrawal</i> activity	104
7.10	The corrected version of the <i>ATM.Withdraw</i> activity	105
8.1	Structure and logic layer of the online Web Shop application	109
8.2	Activity <i>CheckCredentials</i> of the <i>CustomerService</i> class	110
8.3	Activity <i>ConfirmOrder</i> of the <i>OrderService</i> class	112
8.4	Activity <i>GetCart</i> of the <i>OrderService</i> class	114
8.5	Corrected version of the activity <i>OrderService.GetCart</i>	116
A.1	Example of running a test suite	129

## List of Tables

4.1	Modeling concepts of UML activities and actions language included in fUML subset version 1.0 [May11]	47
4.2	UML packages included in fUML subset version 1.0 [May11]	48
8.1	Results of the skills questionnaire	108
8.2	Results of the opinion questionnaire	118
8.3	Comparison of JUnit and the Testing Framework test cases	120

# Listings

6.1	Import statement specification . . . . .	64
6.2	Test scenario specification . . . . .	65
6.3	Test case specification . . . . .	65
6.4	Order assertion specification . . . . .	66
6.5	Order assertion specification with a defined suborder . . . . .	66
6.6	State assertion time frame specification . . . . .	67
6.7	Object state expression specification . . . . .	69
6.8	Property state expression specification . . . . .	69
6.9	Invocation of an OCL constraint in a test case . . . . .	70
6.10	Specification of an OCL constraint . . . . .	70
6.11	Test scenario for testing the <i>Account.MakeWithdrawal</i> activity . . . . .	71
6.12	<i>Account.MakeWithdrawal</i> activity test case: an amount to be withdrawn not exceeding the balance of the account . . . . .	72
6.13	OCL constraints for testing the <i>Account.MakeWithdrawal</i> activity . . . . .	73
6.14	<i>Account.MakeWithdrawal</i> activity test case: an amount to be withdrawn exceeding the balance of the account . . . . .	74
6.15	Test scenario for testing the <i>ATM.Withdraw</i> activity . . . . .	75
6.16	<i>ATM.Withdraw</i> Activity Test Case: an amount to be withdrawn exceeding the balance of the account, with wrong pin provided . . . . .	76
6.17	<i>ATM.Withdraw</i> Activity Test Case: an amount to be withdrawn not exceeding the balance of the account, with a correct pin inserted . . . . .	77
6.18	OCL constraints for testing the <i>ATM.Withdraw</i> activity . . . . .	78
6.19	An example of a terminal rule . . . . .	79
6.20	An example of a rule call . . . . .	79
6.21	Xtext cross-link rule definition . . . . .	80
7.1	Result of the <i>Account.MakeWithdrawal</i> activity test case with exceeding amount . . . . .	102
7.2	Result of the <i>Account.MakeWithdrawal</i> activity test case with non-exceeding amount . . . . .	103
7.3	Result of the <i>ATM.Withdraw</i> activity test case with incorrect pin and exceeding amount . . . . .	104
7.4	Result of the <i>ATM.Withdraw</i> activity test case with correct pin and non-exceeding amount . . . . .	104
8.1	An example test suite implementing requirements for the <i>CheckCredentials</i> activity . . . . .	111
8.2	An example of the test scenario implementing specified requirements for the <i>ConfirmOrder</i> activity . . . . .	112
8.3	An example of the test suite implementing specified requirements for the <i>ConfirmOrder</i> activity . . . . .	113

8.4	OCL constraint for testing the <i>ConfirmOrder</i> activity . . . . .	113
8.5	Test scenario . . . . .	114
8.6	Test cases . . . . .	114
8.7	Test case 1 results report . . . . .	115
8.8	Test case 2 results report . . . . .	115
B.1	Test language grammar declaration . . . . .	131
B.2	Test suite grammar . . . . .	132
B.3	Import statement grammar . . . . .	132
B.4	Test scenario grammar . . . . .	132
B.5	Object specification grammar . . . . .	132
B.6	Attribute specification grammar . . . . .	132
B.7	Simple and object value grammar . . . . .	133
B.8	Link specification grammar . . . . .	133
B.9	Test case grammar . . . . .	133
B.10	Activity input grammar . . . . .	134
B.11	Assertions grammar of the test specification language . . . . .	134
B.12	Order assertion grammar . . . . .	134
B.13	State assertion grammar . . . . .	134
B.14	Temporal operator and temporal quantifier grammar . . . . .	135
B.15	Reference point grammar . . . . .	135
B.16	Grammar specifying possible kinds of state expressions for checking the state in a state assertion . . . . .	135
B.17	Grammar for specifying an invocation of an OCL constraint . . . . .	135
B.18	Grammar for specifying state expressions . . . . .	136
B.19	Arithmetic operator grammar . . . . .	136
B.20	Finally assertion grammar . . . . .	136

# Bibliography

- [ACH<sup>+</sup>13] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based Model Slicing: A Survey. *ACM Computing Surveys*, 45(4):1–36, 2013.
- [AST13] Islam Abdelhalim, Steve Schneider, and Helen Treharne. An Integrated Framework for Checking the Behaviour of fUML Models using CSP. *International Journal on Software Tools for Technology Transfer*, 15(4):375–396, 2013.
- [ATL06] ATLAS Group, LINA & INRIA. ATL: Atlas Transformation Language, User Manual, version 0.7, February 2006. [http://www.eclipse.org/at1/documentation/old/ATL\\_User\\_Manual\[v0.7\].pdf](http://www.eclipse.org/at1/documentation/old/ATL_User_Manual[v0.7].pdf).
- [B05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.
- [Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.
- [BGKS13] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. OCL meets CTL: Towards CTL-Extended OCL Model Checking. In *OCL@MoDELS*, pages 13–22. CEUR-WS.org, 2013.
- [BGL<sup>+</sup>07] Fabrice Bouquet, Christophe Grandpierre, Bruno Legear, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A Subset of Precise UML for Model-based Testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, pages 95–104. ACM, 2007.
- [BVCR06] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML - Towards a System Model for UML. The Structural Data Model, 2006. Technical Report TUM-I0612.
- [BVCR07] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML - Towards a System Model for UML: The Control Model., 2007. Technical Report TUM-I0710.

- [CCGT09] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, 2009.
- [CD08] Michelle L. Crane and Juergen Dingel. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, pages 96–110. ACM, 2008.
- [CML13] Florin Craciun, Simona Motogna, and Ioan Lazar. Towards Better Testing of fUML Models. In *Proceedings of 6th International Conference on Software Testing, Verification and Validation (ICST)*, pages 485–486. IEEE, 2013.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [DD09] Kundu Debasish and Samanta Debasis. A Novel Approach to Generate Test Cases from UML Activity Diagrams. *Journal of Object Technology*, 8(3):65–83, 2009.
- [DeM89] Rich A. DeMillo. Completely Validated Software: Test Adequacy and Program Mutation. In *Proceedings of the 11th International Conference on Software Engineering*, pages 355–356. ACM, 1989.
- [DMC15] Zamira Daw, John Mangino, and Rance Cleaveland. UML-VT: A Formal Verification Environment for UML Activity Diagrams. In *Proceedings of the MoDELS 2015 Demo and Poster Session co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015)*, pages 48–51. CEUR-WS.org, 2015.
- [DTGF<sup>+</sup>05] Trung Dinh-Trong, Sudipto Ghosh, Robert France, Michael Hamilton, and Brent Wilkins. UMLAnT: an Eclipse plugin for animating and testing UML designs. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange*, pages 120–124. ACM, 2005.
- [DTKG<sup>+</sup>05] Trung Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert France, and Anneliese Andrews. A Tool-supported Approach to Testing UML Design Models. In *Proceedings of the 10th IEEE Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 519–528. IEEE Computer Society, 2005.



- [EW04] Rik Eshuis and Roel Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.
- [FJS<sup>+</sup>11] Heidenreich Florian, Johannes Jendrik, Karol Sven, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating OCL and Textual Modelling Languages. In *Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2011.
- [FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of 2007 Future of Software Engineering*, pages 37–54. ACM, 2007.
- [Fra02] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2002.
- [GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by Automatic Snapshot Generation. *Software and System Modeling*, 4(4):386–398, 2005.
- [GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.
- [Gri11] Jurijs Grigorjevs. Model-Driven Testing Approach Based on UML Sequence Diagram. *Scientific Journal of Riga Technical University*, 44:85–90, 2011.
- [Hax10] Anne E Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications, 2010. Technical Report. <http://www2.imm.dtu.dk/courses/02263/F13/Files/FormalMethodsNoteTS.pdf>.
- [HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous Program Slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.
- [HHG14] Frank Hilken, Lars Hamann, and Martin Gogolla. Transformation of UML and OCL Models into Filmstrip Models. In *Theory and Practice of Model Transformations*, volume 8568 of *Lecture Notes in Computer Science*, pages 170–185. Springer International Publishing, 2014.
- [HJK<sup>+</sup>11] Andreas Holzer, Visar Januzaj, Stefan Kugele, Boris Langer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. *Seamless Testing for Models and Code*, pages 278–293. Springer, 2011.

- [HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28(1):75–105, 2004.
- [Hoa78] Charls Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoferssen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011.
- [Int04] International Standards Organization. Process Specification Language, November 2004. ISO 18629.
- [KKBK07] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. Test Cases Generation from UML Activity Diagrams. In *8th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, pages 556–561. IEEE, 2007.
- [KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.
- [LAD<sup>+</sup>14] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, GehanM.K. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, pages 1–38, 2014.
- [LBBG14] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. Formalization of fUML: An Application to Process Verification. In *Advanced Information Systems Engineering*, volume 8484 of *Lecture Notes in Computer Science*, pages 347–363. Springer International Publishing, 2014.
- [LJX<sup>+</sup>04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 284–291. IEEE, 2004.
- [LLP<sup>+</sup>09] Condruț-Lucian Lazăr, Ioan Lazăr, Bazil Pârv, Simona Motogna, and István-Gergely Czibula. Using a fUML Action Language to Construct UML Models. In *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 93–101. IEEE, 2009.

- [LLP<sup>+</sup>10] Condruț-Lucian Lazăr, Ioan Lazăr, Bazil Pârv, Simona Motogna, and István-Gergely Czibula. TOOL SUPPORT FOR FUML MODELS. *International Journal of Computers, Communications and Control*, 5(5):775–780, 2010.
- [LWL08] Bin Lei, Linzhang Wang, and Xuandong Li. UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008*, pages 200–209. IEEE, 2008.
- [May11] Tanja Mayerhofer. Breathing New Life into Models: An Interpreter-Based Approach for Executing UML Models. Master’s thesis, Faculty of Informatics, Vienna University of Technology, 2011. <http://permalink.obvsg.at/AC07810368>.
- [May14] Tanja Mayerhofer. *Defining Executable Modeling Languages with fUML*. PhD thesis, Faculty of Informatics, Vienna University of Technology, 2014. <http://permalink.obvsg.at/AC12132570>.
- [McM92] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.
- [MEBSECdlF05] Beato M. Encarnación, Manuel Barrio-Solórzano, Carlos E. Cuesta, and Pablo de la Fuente. UML Automatic Verification Tool with Formal Methods. *Electronic Notes in Theoretical Computer Science*, 127(4):3–16, 2005.
- [MH03] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., 2003.
- [MLK12] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A Runtime Model for fUML. In *Proceedings of the 7th Workshop on Models@Run.Time*, pages 53–58. ACM, 2012.
- [MLMK13] Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A Framework for Testing UML Activities Based on fUML. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, pages 1–10. CEUR, 2013.
- [MM14] Stefan Mijatov and Tanja Mayerhofer. Challenges of Testing Business Process Models in Intra- and Inter-Organizational Context. In *Joint Proceedings of the 1st International Workshop on Modeling Inter-Organizational Processes and 1st International Workshop on Event*

*Modeling and Processing in Business Process Management co-located with Modellierung 2014, Vienna, Austria*, pages 73–85. CEUR, 2014.

- [MMLK15] Stefan Mijatov, Tanja Mayerhofer, Philip Langer, and Gerti Kappel. Testing Functional Requirements in UML Activity Diagrams. In *Tests and Proofs*, volume 9154 of *Lecture Notes in Computer Science*, pages 173–190. Springer International Publishing, 2015.
- [MXX] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *Proceedings of the 2006 International Workshop on Automation of Software Test, year = 2006, publisher = ACM, pages = 2-8*.
- [Obj11] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0, February 2011. <http://www.omg.org/spec/FUML/1.0>.
- [Obj12] Object Management Group. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012. <http://www.omg.org/spec/OCL/2.3.1>.
- [Obj13a] Object Management Group. Action Language for Foundational UML (Alf), Version 1.0.1, October 2013. = <http://www.omg.org/spec/ALF/1.0.1/>.
- [Obj13b] Object Management Group. UML Testing Profile (UTP), Version 1.2, April 2013. <http://www.omg.org/spec/UTP/1.2/>.
- [Obj14] Object Management Group. Meta Object Facility (MOF) Core, Version 2.4.2, April 2014. <http://www.omg.org/spec/MOF/2.4.2>.
- [Obj15] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5, June 2015. <http://www.omg.org/spec/UML/2.5>.
- [PAK<sup>+</sup>07] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing UML Designs. *Information and Software Technology*, 49(8):892–912, 2007.
- [PAOC13] Matthew Patrick, Robert Alexander, Manuel Oriol, and John A. Clark. Selecting Highly Efficient Sets of Subdomains for Mutation Adequacy. In *20th Asia-Pacific Software Engineering Conference, (APSEC)*, pages 91–98. IEEE, 2013.
- [PCG11] Elena Planas, Jordi Cabot, and Cristina Gómez. Lightweight Verification of Executable Models. In *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 467–475. Springer Berlin Heidelberg, 2011.

- [Rc10] Grigore Roşu and Traian Florin Şerbănută. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
- [Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., 1985.
- [RSGVF14] Alessandro Romero, Klaus Schneider, and Maurício Gonçalves Vieira Ferreira. Using the Base Semantics given by fUML for Verification. In *Proceedings of 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14)*, pages 5–16. SCITEPRESS Digital Library, 2014.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 380–402. Springer Berlin Heidelberg, 2003.
- [RWLN89] Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.
- [RWMR13] Fazle Rabbi, Hao Wang, Wendy MacCaull, and Adrian Rutle. A Model Slicing Method for Workflow Verification. *Electronic Notes in Theoretical Computer Science*, 295(0):79–93, 2013.
- [SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2008.
- [SCWM10] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven Slicing of UML/OCL Models. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 185–194. ACM, 2010.
- [SH05] Harald Störrle and Jan Hendrik Hausmann. Towards a formal semantics of UML 2.0 activities. In *In Proceedings of German Software Engineering Conference, volume P-64 of LNI*, pages 117–128, 2005.
- [SL15] Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.
- [Som06] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 8th edition, 2006.

- [Ste12] Stefan Mijatov. Testing of UML Activity Diagrams, October 2012. Presented at Doctoral Symposium co-located with MODELS 2011 [http://publik.tuwien.ac.at/files/PubDat\\_224358.pdf](http://publik.tuwien.ac.at/files/PubDat_224358.pdf).
- [Stö04a] Harald Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 235–242. IEEE, 2004.
- [Stö04b] Harald Störrle. Semantics of Exceptions in UML 2.0 Activities, 2004. Technical report.
- [Stö04c] Harald Störrle. Semantics of Structured Nodes in UML 2.0 Activities. In *2nd Nordic Workshop on UML Modeling, Methods and Tools (NWUML)*, 2004.
- [Stö05] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52, 2005.
- [Tea02] CMMI Product Team. CMMI for Software Engineering, Version 1.1, Staged Representation (CMMI-SW, V1.1, Staged), 2002.
- [Utt05] Mark Utting. The Role of Model-Based Testing. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005*, pages 510–517. Springer, 2005.
- [Wac08] Guido Wachsmuth. Modelling the Operational Semantics of Domain-Specific Modelling Languages. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 506–520. Springer Berlin Heidelberg, 2008.
- [Wu06] Hui Wu. Grammar-driven Generation of Domain-specific Language Tools. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 772–773. ACM, 2006.
- [xte13] Xtext 2.5 Documentation, December 2013. <http://eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf>.
- [ZFKB12] Philipp Zech, Michael Felderer, Philipp Kalb, and Ruth Breu. A Generic Platform for Model-Based Regression Testing. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin Heidelberg, 2012.

# Curriculum Vitae

## Basic Info

**Dipl.-Ing. Stefan Mijatov, MSc**

**Address:** 1045 Wellington Street, Montreal QC, Canada

**Email:** stefan.mijatov@gmail.com

**Date of birth:** March 6th, 1985

## Educational Background

### *PhD Student*

Faculty of Informatics, TU Vienna, Austria

2011 - 2018

### *Master of Science Degree*

Faculty of Organizational Sciences, University of Belgrade, Serbia

2010 - 2011

### *Bachelor of Science Degree*

Faculty of Organizational Sciences, University of Belgrade, Serbia

2004 - 2009

### *Electrical Engineering High School*

'Nikola Tesla', Belgrade, Serbia

2000 - 2004

## Professional Background

### *Associate in the Operations and Compliance Technology Division*

*Morgan Stanley Services Canada Corp, Montreal, QC*

April 2017 - Present

Since April 2017, I am employed as Associate in the Operations and Compliance Technology Division of the Morgan Stanley Services Canada Corp. In this position, I am involved in software development of financial and trade surveillance analysis models.

The models are based on Apache Spark technology for utilizing computer clusters for analyzing big volumes of data, where statistical functions are applied for analysis and surveillance purposes.

***Senior Software Developer***  
***IOCS Systems / DealFlo, Montreal, QC***  
April 2015 - March 2017

Since April 2015, I was employed as a senior software developer in IOCS Systems, a start-up company providing a SaaS platform for execution of high value financial transactions involving production of secure agreements with electronic signatures, evidence recording in form of the complete trace-ability of the signing process, and secure storage of signed agreements.

Among several projects I worked on, I was assigned as a technical lead in the project of re-factoring the core of the IOCS platform. During the work on this project, the amount of the code base have been significantly reduced, and by applying standard software design patterns the internal structure of the system has been widely improved, increasing the ability of the development team to more effectively and efficiently execute on client implementation and road-map projects.

***Software Developer***  
***iNova, Ltd., Belgrade, Serbia***  
February 2011 - October 2011

Two months before graduating MSc degree, I was employed as a software developer in a small company that develops Information Systems and Geographical Information Systems. I was involved in design and development of an Information System for Investment and Material Management, built using .NET platform.

## **Publications**

- S. Mijatov, T. Mayerhofer, P. Langer, G. Kappel, Testing Functional Requirements in UML Activity Diagrams  
in: Proceeding of the Tests and Proofs: 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015.
- S. Mijatov, T. Mayerhofer, Challenges of Testing Business Process Models in Intra- and Inter- Organizational Context  
Talk: 1st International Workshop on Modeling Inter-Organizational Processes, MinoPro 2014 @ Modellirung 2014; Vienna; 03-19-2014
- S. Mijatov, P. Langer, T. Mayerhofer, G. Kappel, A Framework for Testing UML Activities based on fUML  
Talk: 10th International Workshop on Model Driven Engineering, Verification and Validation, Miami; 10-01-2013; in: Proceedings of the 10th International Workshop



on Model Driven Engineering, Verification and Validation co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), CEUR, Vol-1069 (2013), ISSN: 1613-0073; 1 - 10.

- S. Mijatov, Testing of UML activity diagrams  
Talk: Doctoral Symposium @ ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems MODELS 2012, Innsbruck; 10-02-2012.