FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Defining Executable Modeling Languages with fUML

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Sozial- und Wirtschaftswissenschaften

eingereicht von

Tanja Mayerhofer
Matrikelnummer 0625154

an der
Fakultät für Informatik der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

(O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel)	(Ed Seidewitz)

Wien, 17.11.2014

(Tanja Mayerhofer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Defining Executable Modeling Languages with fUML

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Sozial- und Wirtschaftswissenschaften

by

Tanja Mayerhofer
Registration Number 0625154

to the Faculty of Informatics
at the Vienna University of Technology

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

The dissertation has been reviewed by:

(O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel)	(Ed Seidewitz)

Wien, 17.11.2014

(Tanja Mayerhofer)

Erklärung zur Verfassung der Arbeit

Tanja Mayerhofer
Richtergasse 1a/5, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgments

Many people have contributed to the successful completion of my dissertation. At this point, I want to say thank you to everyone who has supported me on my way. In particular, I want to thank the following people for their encouragement and advice.

Thank you Gerti for giving me the opportunity to do my dissertation in your research group and supporting me in any organizational, strategical, and research-related matter.

Thank you Ed for providing me with many insights into the modeling languages we both like so much and your valuable feedback on my research.

Thank you Daniel and Peter from LieberLieber Software GmbH for partly funding my research.

Thank you Philip for taking care of me the last three years and encouraging me to carry on with my research. Without you I would not have written this thesis.

Thank you Manuel for bringing me to this research group in the first place and spending many hours with me to discuss my research.

Thank you Alex, Christian, Dieter, Javier, Katja, Konrad, Luca, Manuel, Martin, Patrick, Philip, and Stefan for contributing to the nice working environment I enjoyed during the last three years.

Thank you Mama, Papa, Conny, Carina, and Johannes for your love and support.

Abstract

Model-driven engineering (MDE) is a software development paradigm aiming to cope with the growing complexity of software systems by raising the level of abstraction. In this paradigm, a system's structure and behavior are defined by means of models using modeling languages that enable developers to abstract away from implementation and platform details. From the defined models, complete software system implementations may be (semi-)automatically generated by utilizing model transformation and code generation techniques.

As MDE puts models into the center of software development, adequate methods and techniques for creating, analyzing, and utilizing models constitute a crucial prerequisite for the successful adoption of MDE. Due to the large body of modeling languages used in MDE including general purpose and domain specific languages, means for efficiently developing adequate tool support for modeling languages are needed. To address this need, the automation techniques provided by MDE may also be applied to automate the development of tool support for modeling languages. This is current practice for developing syntax-based tools, such as modeling editors. Thereby, syntax-based tools are generated from the definition of a modeling language's syntax specified by means of standardized metamodeling languages. In contrast, the automated development of semantics-based tools, such as model debuggers, has not reached the same level of maturity yet. Partly, this is due to the lack of a standardized language for defining the semantics of modeling languages.

The goal of this thesis is to fill this gap and provide a solution for automating the development of semantics-based tools for executable modeling languages based on behavioral semantics specifications. The *first contribution* of this thesis comprises a language and methodology for developing behavioral semantics specifications based on the standardized language fUML. fUML is an executable subset of UML providing a formal semantics and execution environment. The *second contribution* of this thesis comprises extensions of fUML's execution environment with means for execution control, runtime observation, and runtime analysis building the basis for developing semantics-based tools. Based on these extensions, the *third contribution* of this thesis provides a generic model execution environment for modeling languages whose behavioral semantics is defined with fUML. The *fourth contribution* consists in a generic semantic model differencing framework, which builds upon the first three contributions and takes—in contrast to existing model differencing approaches—the behavioral semantics of models into account for reasoning about differences among models. With these contributions, we aim at providing a stimulus towards the establishment of a common behavioral semantics specification language in MDE, as well as laying the basis for future innovations regarding the automated development of semantics-based tools for executable modeling languages.

Kurzfassung

Model-Driven Engineering (MDE) ist ein Paradigma in der Softwareentwicklung, das die Verwendung von Softwaremodellen vorsieht, um ein höheres Abstraktionsniveau im Entwicklungsprozess zu erreichen und dadurch die Entwicklung komplexer Softwaresysteme zu erleichtern. Dazu werden Softwaremodelle mithilfe von Modellierungssprachen erstellt, die es erlauben, die Struktur und das Verhalten des zu entwickelnden Softwaresystems auf einer geeigneten Abstraktionsebene zu definieren. Die lauffähige Software wird aus den Softwaremodellen mittels Modelltransformationstechniken (teil-)automatisiert abgeleitet.

In MDE stellen Softwaremodelle die zentralen Entwicklungsartefakte dar, wodurch Techniken zur Erstellung, Analyse und Weiterverarbeitung von Softwaremodellen eine entscheidende Rolle im Entwicklungsprozess zukommt. Aufgrund der Vielzahl an eingesetzten Modellierungssprachen bedarf es auch effizienter Methoden zur Entwicklung entsprechender Modellierungswerkzeuge. Zu diesem Zweck können die Techniken, die in MDE verwendet werden, um Softwaresysteme effizient zu entwickeln, auch zur effizienten Entwicklung von Modellierungswerkzeugen eingesetzt werden. Das ist gängige Praxis in der Entwicklung syntax-basierter Werkzeuge, wie etwa Editoren. Diese werden aus der Syntaxdefinition einer Modellierungssprache automatisiert abgeleitet, wobei die Syntax mittels standardisierten Sprachen spezifiziert wird. Im Gegensatz dazu ist die automatisierte Entwicklung semantik-basierter Werkzeuge, wie Simulationsumgebungen, bisher nicht im gleichen Maß möglich. Ein Grund dafür ist das Fehlen einer standardisierten Sprache zur Definition der Semantik von Modellierungssprachen.

Das Ziel dieser Arbeit ist, diese Lücke zu füllen und einen Lösungsansatz für die automatisierte Entwicklung von semantik-basierten Modellierungswerkzeugen bereitzustellen. Dazu wird in dieser Arbeit eine Sprache zur Definition der Semantik von Modellierungssprachen vorgestellt, die auf der standardisierten Sprache fUML aufbaut. Der Sprachumfang von fUML umfasst einen Teil der weitverbreiteten Modellierungssprache UML. fUML verfügt über eine formale Semantikdefinition sowie eine Ausführungsumgebung. Um die Basis für die Entwicklung semantik-basierter Modellierungswerkzeuge zu schaffen, wird in dieser Arbeit die Ausführungsumgebung von fUML um Kontroll-, Überwachungs- und Analysemechanismen erweitert. Basierend auf diesen Erweiterungen wird eine generische Ausführungsumgebung vorgestellt, die für jede Modellierungssprache verwendet werden kann, deren Semantik mit fUML definiert ist. Diese generische Ausführungsumgebung bildet u.a. die Grundlage für den semantischen Vergleich von Modellen. Die vorliegende Dissertation ist impulsgebend für das Schaffen einer gemeinsamen Sprache zur Definition der Semantik von Modellierungssprachen, und damit Basis für zukünftige Innovationen in der automatisierten Entwicklung von semantik-basierten Modellierungswerkzeugen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Since the early days of software engineering, *abstraction* is a key enabler for coping with the growing complexity of the software systems that have to be built. Abstraction is the process of focusing on certain details about an object, which are relevant for a specific purpose, and hiding details, which are not relevant for this specific purpose. Thereby, abstraction enables the reduction of complexity and, hence, the design and implementation of complex software systems. For instance, the *language abstraction* [57, 137] achieved by the transition from assembly languages to third-generation programming languages, which started in the late 1950s, enables programmers to develop software independently of the underlying used machine, as compilers and interpreters take care of machine-dependent concerns. Likewise, *platform abstractions* [57, 137] provided by reusable software libraries and software frameworks, such as JEE or .NET, allow developers abstracting away from platform technology details.

Language abstractions and platform abstractions achieved so far in software engineering provide abstractions for the *solution space*, i.e., they provide abstractions for the underlying computing technologies. Model-driven engineering (MDE) seeks to go one step further and raise the level of abstraction beyond programming languages and platforms. Instead of specifying the solution for a problem using low-level programming language concepts and specific platform concepts, domain concepts closer to the *problem space* are used to express the system design [16, 57, 137, 139, 158]. Therefore, modeling languages are used to define the requirements, the structure, and the behavior of the software system to be built formally[1] in terms of models, which can be further processed by computers to automatically generate software system implementations for multiple platforms. For this automation, transformation engines and code generators are used, which act as platform-specific compilers for the models and synthesize various types of software artifacts, such as source code, database schemata, and deployment scripts.

[1] In this thesis, the term "formal" is used interchangeably with the term "computer processable" if not explicitly stated otherwise. This definition should not be confused with the usage of the term "formal" in mathematics or logics.

Hence, the models constitute the design, implementation, and documentation of the software system at the same time. Due to the achieved automation, MDE supports not only handling the complexity of the software systems to be built, but also enables increasing the productivity of software development.

The application of MDE induces a shift from code-centric software development to *model-centric software development*. Thus, models constitute the central artifacts in software development and serve as the single specification of the system to be built. As a consequence, the success of MDE depends significantly on the availability of adequate tool-supported methods and techniques for creating, manipulating, exploring, analyzing, and utilizing models. In particular, methods and techniques supporting the development of high-quality models, such as model debugging, model testing, and dynamic model analysis are crucial in MDE. Tools implementing such methods and techniques are specific to the modeling language in use. While on the one hand, general purpose modeling languages, such as UML, are used in MDE, MDE also promotes the development and usage of domain specific modeling languages. They are designed specifically for a certain domain to ease the development of models capturing the problem domain of the system to be built. Hence, methods, techniques, and tools for processing models defined with a variety of distinct modeling languages are a crucial prerequisite for the successful adoption of MDE.

1.2 Problem Statement

Developing distinct tools for each modeling language in use manually and from scratch is impractical. This is an apparent issue, especially if we take into account that a large body of distinct modeling languages is available and that similar tool capabilities, such as model editing, model execution, model debugging, and model testing, are required for all these modeling languages. Fortunately, the techniques available in MDE to automate the development of software systems can be applied to also automate the development of tool support for modeling languages [20]. To accomplish this, modeling languages have to be defined formally, meaning that their syntax as well as their semantics have to be defined in a computer processable way. Such a formal definition of a modeling language can be processed using MDE techniques to provide specific tool support for a modeling language.

The *syntax* of a modeling language defines the modeling language's concepts and the relations between these concepts. Thus, it defines the structure of models which can be constructed with the modeling language. The computer internal representation of the syntax of a modeling language is also referred to as *abstract syntax*. For formally defining the abstract syntax of a modeling language, metamodels constitute the standard means. The MOF standard [121], developed by the Object Management Group (OMG), constitutes a standardized and well established metamodeling language for this purpose. Furthermore, MOF laid the ground for automating the development of a variety tools that build upon the abstract syntax definition of a modeling language. In particular, it led to the availability of techniques for (semi-)automatically deriving modeling editors from a metamodel and generic components for model serialization, comparison, and transformation.

The *semantics* of a modeling language defines the meaning of each concept provided by the language and, hence, defines how models conforming to the language's abstract syntax definition have to be interpreted [63]. A formal definition of a modeling language's semantics is not only needed for precisely and unambiguously defining the meaning of conforming models, but also to establish the basis for an efficient development of tools building upon the semantics of a modeling language, such as model execution engines, model debuggers, and model testing environments. Having the semantics of a modeling language explicitly and formally defined enables the automation of the development of such tools. Unfortunately, no standard way for formally defining the semantics of a modeling language has been established yet. In fact, the semantics of many modeling languages, including widely adopted languages, such as UML[2] [113], are only informally defined in natural language. The lack of a standardized language for formally defining the semantics of modeling languages impedes the efficient or even automated development of semantics-based tools. Thus, formalizing the semantics of modeling languages remains a core challenge in MDE [20]. Having a standardized and well established semantics specification language in MDE may provide similar benefits as MOF granted for developing tools building upon a modeling language's abstract syntax. In particular, it may enable the emergence of (semi-)automatic derivation techniques and reusable generic components for developing tools which build upon a modeling language's semantics.

This thesis is concerned with the challenge of formally and explicitly defining the semantics of modeling languages and efficiently developing semantics-based tools on top of the definition of a modeling language's semantics. In particular, we focus on *executable modeling languages* that allow the definition and analysis of dynamic aspects of systems by means of executable models. Executable models enable the analysis of software systems starting from the early phases of the development process. Therefore dedicated analysis tools may be used that implement analysis methods and techniques supporting understanding, exploring, validating, and verifying models. Examples of such analysis methods are debugging, testing, and dynamic analysis methods. However, to develop analysis tools for executable modeling languages in an efficient manner, their semantics has to be formally defined comprising the formal definition of the execution behavior of conforming models. This kind of semantics is referred to as *behavioral semantics*.

As pointed out above, no standard way for formally defining the semantics of modeling languages exists. This is also true for behavioral semantics. As will be discussed in Section 2.2.2, several approaches for defining the semantics of modeling languages exist. However, none of these approaches are widely adopted, especially compared to the wide adoption of metamodeling languages, such as MOF. Furthermore, only a few existing approaches address the challenge of automating the development of semantics-based tools, and these approaches provide only partial solutions for this issue [20].

Due to these deficiencies, the behavioral semantics of many modeling languages is only informally defined in natural language. This may lead to several problems, such as ambiguities

[2]The semantics of a subset of UML, called the foundational UML (fUML) subset, has been formally defined and standardized by the OMG in 2011 [114]. As this thesis builds heavily upon fUML, it will be described in detail in Section 3. However, the semantics of the concepts of UML, which are not included in the fUML subset, is only described in English prose in the UML standard [113].

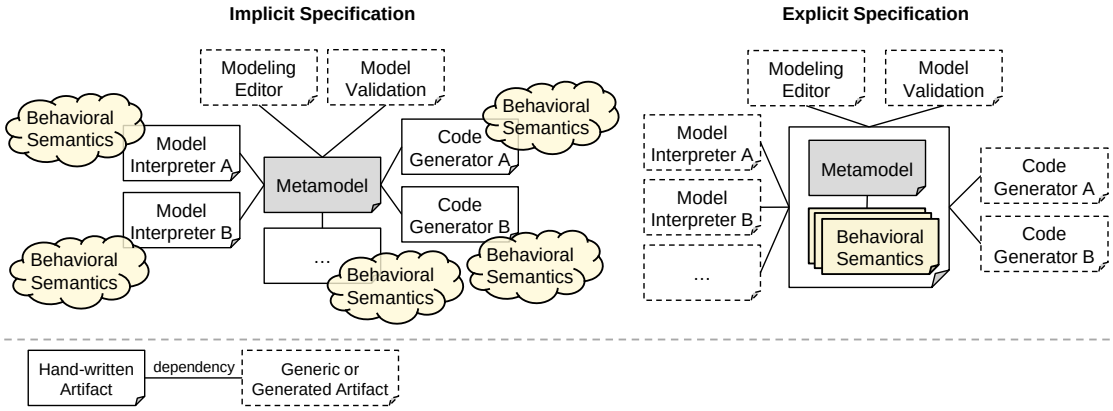


Figure 1.1: Comparison of implicit and explicit specification of behavioral semantics

in the semantics of the languages' concepts. Furthermore, the behavior of models cannot be formally analyzed and the models cannot be executed. Moreover, the development of tools, such as model debuggers and model testing environments, is problematic if the used language's behavioral semantics is only defined in natural language.

A commonly used approach to make models executable is to develop code generators or model interpreters with general purpose programming languages (cf. left-hand side of Figure 1.1). However, code generators or model interpreters constitute only *implementations* of the behavioral semantics rather than explicit *specifications*. In this approach the behavioral semantics is encoded in the manually constructed code generation templates or model interpreter implementations and, hence, is only defined implicitly. Furthermore, the behavioral semantics might be redundantly defined, e.g., in distinct code generators for distinct target platforms, and it might be only partially defined, as, e.g., specific code generators might only consider certain selected concepts of a modeling language. Thus, it is difficult to analyze, extend, and reuse the implemented semantics, as well as to verify whether the implementations are actually consistent with each other regarding the intended semantics, making it costly to create and maintain such implementations.

To overcome these limitations, a standardized way for formally and explicitly specifying the behavioral semantics of modeling languages is needed (cf. right-hand side of Figure 1.1). Moreover, a model-based specification of the behavioral semantics would be beneficial because it enables staying in the technical space of MDE and, hence, immediately applying MDE techniques for processing such specifications. In particular, a formal, explicit, and model-based behavioral semantics specification could be processed utilizing MDE techniques for automating the development of semantics-based tools, such as generating model execution facilities from behavioral semantics specifications or developing generic model execution facilities operating on behavioral semantics specifications.

1.3 Aim of the Work

This thesis aims at addressing the challenge of formally and explicitly specifying the behavioral semantics of executable modeling languages and enabling an efficient development of semantics-based tools for modeling languages on top of their semantics specifications. In particular, we investigate how the standardized UML 2 compliant action language of *foundational UML (fUML)* [114] can be used as a semantics specification language in MDE and how semantics specifications developed with this language can be used to efficiently develop semantics-based tools for modeling languages.

The fUML standard published by OMG formally defines the behavioral semantics of a subset of UML, which is called the *foundational UML subset*. The behavioral semantics of the fUML subset is defined in terms of a virtual machine, which enables the execution of models compliant to this subset. The fUML subset consists of UML modeling concepts for defining the structure of a system with UML classes and the behavior of UML classes with UML activities making use of UML's action language. Because UML classes and MOF metaclasses differ only in their intended usage, that is modeling of systems and modeling of languages, respectively, fUML's action language might be well suited not only for specifying the behavior of UML classes, but also for specifying the behavior of MOF metaclasses. This would enable the formal definition of the behavioral semantics of modeling languages in terms of UML activities. As fUML is an object-oriented and imperative action language, well known in the MDE community as it is a subset of UML, and like MOF standardized by OMG, fUML may be considered as a promising candidate for becoming a standardized semantics specification language in MDE.

Although OMG considers fUML as being sufficient for specifying the semantics of the remainder of UML [114, p. 19] and, hence, enable the execution of any UML conform model using the fUML virtual machine, it is, however, an open question how fUML can be employed for this purpose. Moreover, it has not been investigated so far how fUML can be integrated with state of the art metamodeling languages, metamodeling methodologies, and metamodeling environments for providing the means to specify the semantics of other modeling languages in a way that enables the direct execution of models and the development of semantics-based tools relying on this model execution capability. In particular, the following research questions have been addressed by this thesis.

(i) How can fUML be integrated with existing metamodeling languages, methodologies, and environments to be usable for specifying the behavioral semantics of modeling languages in a systematic and efficient manner?

(ii) How can fUML's execution environment be utilized for executing models based on a modeling language's behavioral semantics specification defined with fUML?

(iii) How can semantics-based tools be efficiently developed based on a modeling language's behavioral semantics specification defined with fUML?

These research questions have been addressed by this thesis resulting in four contributions, which are depicted in Figure 1.2. Figure 1.2 depicts three basic phases of an MDE process. In the *language design* phase, the modeling language to be used in the software development

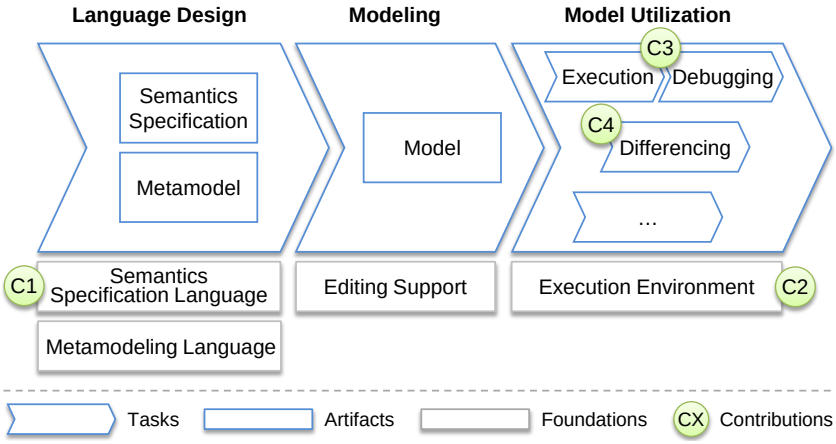


Figure 1.2: Contributions of this thesis

process is defined, which comprises the specification of the modeling language's abstract syntax by means of a metamodel, as well as the specification of its semantics. Contribution 1 of this thesis is concerned with the specification of the behavioral semantics of a modeling language based on fUML. In the *modeling* phase, models conforming to the developed modeling language are created using available editing support. The *model utilization* phase constitutes the phase of using the models for specific purposes by employing dedicated tools. This thesis focuses on utilizing the behavior of models, which is defined by the used modeling language's behavioral semantics. The contributions 2, 3, and 4 of this thesis are concerned with utilizing the executability of models, in particular for model execution, model debugging, and semantic model differencing. The contributions are described in further detail in the following.

Contribution 1: Semantics specification with fUML. To make fUML usable as a semantics specification language and, hence, enable the definition of the behavioral semantics of modeling languages with fUML, fUML has to be integrated with existing metamodeling languages, metamodeling methodologies, and metamodeling environments. The first contribution of this thesis comprises a strategy for integrating the standardized action language of fUML with existing metamodeling languages. The language resulting from this integration constitutes an executable metamodeling language that enables not only to define the abstract syntax of a modeling language, but also to define a modeling language's behavioral semantics. Furthermore, this contribution comprises a methodology for systematically and efficiently developing behavioral semantics specifications with this executable metamodeling language. This methodology seamlessly integrates with existing metamodeling methodologies and metamodeling environments. We have instantiated this strategy and methodology for the Eclipse Modeling Framework (EMF) [150]. In particular, we have integrated fUML's action language with EMF's metamodeling language Ecore and implemented prototypical tool support for developing behavioral semantics specifications according to the elaborated methodology. This contribution addresses the research question *(i)*.

Contribution 2: Extensions of the fUML execution environment. The fUML virtual machine enables the execution of UML models, which conform to the UML subset considered by fUML. To execute a UML model, one has to provide the UML activity that shall be executed as well as input parameter values for this activity to the fUML virtual machine. The output of the execution comprises the output parameter values obtained by the execution. This output can be used for further processing the execution result of the executed UML activity. For instance, using this output, it can be asserted whether the output parameter values correspond to the expected values for a defined input. However, the fUML virtual machine lacks in providing means for execution control, runtime observation, and runtime analysis. This hinders the development of tools which build upon these capabilities of a virtual machine. For instance, model debuggers rely on the capability to control the execution, i.e., to suspend an ongoing execution and to resume the execution stepwise, as well as on the capability to observe the execution, i.e., to retrieve the current state of the execution comprising the next model element to be executed, as well as the currently existing values. Furthermore, for analyzing the execution of a UML activity a posteriori after the execution finished, e.g., for verifying the functional correctness of the UML activity through testing, an execution trace is required, which captures the runtime behavior of the executed UML activity. To overcome these limitations of the fUML virtual machine, the second contribution of this thesis consists in extensions of the fUML virtual machine comprising a command interface for enabling execution control, an event mechanism for enabling runtime observation, and a trace model for enabling runtime analysis. This contribution addresses the research question *(ii)*.

Contribution 3: Semantics-based tool development. Using fUML as a semantics specification language enables the utilization of fUML's execution environment, i.e., the fUML virtual machine, as a generic model execution environment for any modeling language whose behavioral semantics is specified with fUML. This generic model execution environment takes as input a model to be executed as well as the fUML-based semantics specification of the modeling language the model conforms to, and executes the model according to the semantics specification by leveraging the fUML virtual machine. Therefore, the model to be executed is provided as input to the fUML virtual machine, which executes the fUML-based semantics specification for this input. On top of this generic model execution environment, semantics-based tools can be developed, which implement methods and techniques enabling the utilization of models. We have shown how this is possible by developing prototypical implementations of dedicated semantics-based tools. In particular, we have developed a model execution tool, which provides the output of a model execution to the user via annotations of the executed model, as well as a model debugger, which enables the stepwise execution of a model and the inspection of the current state of the execution again via annotations of the executed model. This contribution addresses the research question *(iii)*.

Contribution 4: Semantic model differencing. The fourth contribution of this thesis constitutes an approach for differencing models based on their semantics, i.e., based on their behavior. The majority of existing model differencing approaches compare models only based on their abstract syntax representation, but do not take their semantics into account. Hence, we consider

our semantic model differencing approach as a contribution on its own. The identification of differences among independently developed or consecutive versions of models is not only a crucial prerequisite for several important development and change management tasks, such as model merging and incremental testing, but also for enabling developers to efficiently comprehend a model's evolution. The majority of existing model differencing approaches use a syntactic differencing approach, which applies a fine-grained comparison of models based on their abstract syntax representation. Although syntactic differences constitute valuable and efficiently processable information sufficient for several application domains, they are only an approximation of the semantic differences among models with respect to their meaning. In fact, a few syntactic differences among models may induce considerable semantic differences, whereas syntactically different models may still induce the same semantics. We have developed a generic semantic model differencing framework that can be instantiated to realize semantic differencing operators for specific modeling languages. This generic semantic model differencing framework utilizes the specification of the behavioral semantics of a modeling language to support the semantic differencing of models. In particular, it exploits the executability of models, which is provided by the behavioral semantics specification of the used modeling language, to obtain execution traces for the models to be compared. These execution traces constitute semantic interpretations of the models and, thus, act as input to the semantic comparison. The actual comparison logic can be specified in terms of dedicated match rules defining which syntactic differences among these interpretations constitute semantic differences among the models. This contribution addresses the research question *(iii)*.

Implementations. The artifacts developed in the course of this thesis have been realized as research prototypes. These prototypes are integrated with EMF and published under the Eclipse Public License Version 1.0[3]. Appendix B provides further information about the prototypes.

1.4 Methodological Approach

For carrying out this thesis, the design science paradigm has been applied as the methodological approach. Design science is a constructive methodological approach where knowledge is created by building and evaluating innovative artifacts. Hevner *et al.* [67, 68] introduced a conceptual framework as well as seven guidelines for applying design science in information systems research. These seven guidelines have been applied in this thesis as described in the following.

1. Design as an artifact. The aim of this thesis is to design an approach for formally specifying the behavioral semantics of modeling languages with fUML and to enable the efficient development of semantics-based tools. More precisely, the artifacts described in the following have been built in the course of this thesis.

1. A strategy for integrating fUML with existing metamodeling languages to enable its usage as a semantics specification language and an instantiation of this strategy for the metamodeling language Ecore.

[3] `http://www.eclipse.org/legal/epl-v10.html`, accessed 11.09.2014

2. A methodology for systematically and efficiently developing executable semantics specifications with the elaborated fUML-based semantics specification language and an instantiation of this methodology in the form of tool support for EMF.

3. Extensions of fUML's execution environment providing means for execution control, runtime observation, and runtime analysis.

4. A generic model execution environment based on the extended fUML execution environment enabling the execution of models, which conform to any modeling language whose behavioral semantics is specified with the fUML-based semantics specification language.

5. Implementations of semantics-based tool prototypes based on the generic model execution environment, in particular, a model execution tool as well as a model debugger.

6. A semantic model differencing framework and an instantiation of this framework for fUML-based behavioral semantics specifications, which is based on the generic model execution environment.

2. Problem relevance. Formalizing the behavioral semantics of modeling languages and leveraging this formalization for efficiently developing semantics-based tools is an open challenge in MDE [20]. Despite the fact that several approaches for formally and explicitly specifying the behavioral semantics of modeling languages exist, none of these approaches is widely adopted when compared to the adoption of metamodeling languages. Moreover, the challenge of efficiently developing tools for modeling languages based on their semantics specifications is only addressed by a few approaches which provide only partial solutions [20]. This thesis addresses these challenges by providing a language for specifying the behavioral semantics of modeling languages as well as a generic model execution environment enabling the efficient development of semantics-based tools, which are both based on the standardized action language of fUML.

3. Design evaluation. The artifacts developed in the course of this thesis have been evaluated by applying them to selected modeling languages in well defined case study setups. In particular, the developed semantics specification language as well as the tool support for the elaborated semantics specification methodology have been applied to develop the behavioral semantics specifications of dedicated modeling languages. For the same modeling languages, the developed semantics-based tool prototypes have been applied. To evaluate the extensions of the fUML execution environment, tools that depend heavily on these extensions have been developed. In particular, we have developed a debugger, a testing framework, as well as a tool for performing non-functional property analysis for fUML models. Furthermore, we have evaluated the overhead caused by these extensions regarding execution time and memory consumption. The conducted case studies have been used to draw conclusions about the general applicability of the developed artifacts and they have provided feedback for further refining these artifacts.

4. Research contributions. The presented artifacts and the conclusions drawn from their evaluation constitute the contributions of this thesis to the knowledge base of the MDE community. For specifying the behavioral semantics of modeling languages, various approaches have been proposed in the past, which are, however, not widely adopted. With the approach of specifying the behavioral semantics of modeling languages using the standardized action language of fUML we aim to provide a stimulus towards the establishment of a common behavioral semantics specification language in MDE. Furthermore, the need for efficiently developing semantics-based tools has gotten more attention recently in the research community [20]. This thesis contributes an approach for developing semantics-based tools on top of a generic model execution environment, which is itself based on the fUML virtual machine. This contribution aims to lay the basis for future innovations regarding the automated development of semantics-based tools for modeling languages. Furthermore, we have developed an approach for semantic model differencing to contribute a solution to the MDE research area concerned with model evolution. Moreover, the extensions of the fUML execution environment regarding execution control, runtime observation, and runtime analysis contribute to the capabilities of the standardized fUML execution environment and might provide an input for further enhancing the fUML standard.

5. Research rigor. Existing work in the area of specifying the behavioral semantics of modeling languages as well as automating the development of semantics-based tools has been taken into account during building and evaluating the research artifacts of this thesis. Therefore, intensive literature studies have been carried out prior to designing the artifacts and existing approaches and tools have been reviewed. Furthermore, evaluation methods applied by existing related work have been investigated for their applicability in evaluating the artifacts of this thesis. Moreover, the artifacts built in the course of this thesis have been also contrasted and compared with the artifacts built in related work.

6. Design as a search process. The artifacts developed in this thesis have been iteratively built and evaluated. These iterations have been example-driven. This means that the artifacts have been first designed and built to support simple exemplary modeling languages, and they have been evaluated by conducting case studies applying the artifacts to such modeling languages. By doing so, design alternatives for the artifacts have been explored, the problem domain has become better understood, and experience has been gained. Hence, more complex modeling languages have been considered subsequently and the artifacts have been further refined.

7. Communication of research. The contributions of this thesis have been communicated through well known and peer reviewed publication venues in the MDE community as well as in the broader software engineering community.

1.5 Structure of the Work

This thesis is structured according to the elaborated contributions. In the following, an overview of this thesis is provided by briefly describing the contents of each chapter. Some of the contributions of this thesis have already been published in peer reviewed workshops and conferences.

Hence, the contents of these publications overlap with the contents of this thesis. The following chapter overview provides also information about which contents have already been published.

Chapter 2: State of the Art. In this chapter, we first provide an overview of methods and techniques utilizing the executability of modeling languages for supporting the comprehension, exploration, validation, and verification of models. Thereafter, we provide an introduction into how modeling languages are defined focusing on metamodeling and semantics specification. Furthermore, we briefly introduce executable UML and in particular the fUML standard.

Chapter 3: Foundational UML. In this chapter, we provide a thorough description of the fUML standard. In particular, we introduce the UML modeling concepts addressed by the fUML standard as well as the fUML virtual machine enabling the execution of fUML conform models.

Chapter 4: Extensions of the fUML Execution Environment. This chapter deals with contribution 2. We discuss current limitations of the fUML virtual machine as defined by the fUML standard concerning execution control, runtime observation, and runtime analysis. Thereafter, we present the extensions of the fUML virtual machine, which have been developed in order to overcome these limitations comprising an event mechanism, command interface, and trace model. The extensions of the fUML virtual machine have been published in [97].

Chapter 5: Semantics Specification with fUML. This chapter deals with the contributions 1 and 3. It is concerned with the integration of fUML with existing metamodeling languages, methodologies, and environments enabling the formal definition of the behavioral semantics of modeling languages based on fUML. Furthermore, we present the generic model execution environment built on top of the extended fUML execution environment that provides means for executing models according to fUML-based behavioral semantics specifications as well as the basis for efficiently developing semantics-based tools. The content of this chapter has been published in [98–100].

Chapter 6: Semantic Model Differencing. This chapter deals with contribution 4. We present our generic semantic model differencing framework, which utilizes the behavioral semantics specification of a modeling language to identify semantic differences among models. In particular, this approach makes use of the possibility to execute the models to be compared based on the behavioral semantics specification of the used modeling language. This semantic model differencing framework has been published in [85, 86].

Chapter 7: Evaluation. In this chapter, the evaluation of the developed artifacts as well as the results of this evaluation are presented. The extensions of the fUML execution environment, the semantics specification approach based on fUML, as well as the semantic model differencing approach have been evaluated separately in dedicated case studies. Parts of these case studies have been published in [10, 49, 86, 97, 100, 104, 105].

Chapter 8: Conclusion and Future Work. In this chapter, the contributions of this thesis are summarized, overall conclusions are discussed, and limitations to be addressed in future work are pointed out.

CHAPTER 2

State of the Art

2.1 Executable Modeling Languages and Their Applications

With the application of MDE, software development is centered around models constituting the single specification of the software systems to be built. Therefore, modeling languages are used, which enable the specification of the structure and the behavior of software systems formally and completely on a higher level of abstraction and closer to the problem space than possible with existing programming languages. Thereby, the goal of applying MDE is on the one hand to cope with the growing complexity of the software systems to be built and on the other hand to improve the productivity and quality of software development [16, 57, 137, 139, 158]. To achieve this, high-level and abstract models are created and refined manually or automated through model transformations, until models detailed enough to automatically generate complete software system implementations are obtained. This automation step is achieved by code generators, which act as platform-specific compilers transforming the models into executable software artifacts, in particular into source code executable on the intended target platform.

For this purpose, *executable modeling languages* may be used. Executable modeling languages are modeling languages that enable not only the specification of the static aspects of a system, but also the dynamic aspects, i.e., the behavior of the system, by means of *executable models* [17]. Thereby, a model is executable, if *"it is possible to write an engine program that executes (or runs) the model"* [73, p. 7]. The main advantage of executable modeling languages is that they can be used to formally specify a software system. This implies that the meaning of an executable model is precisely defined, and hence, executable models can be processed by computers. On the one hand, executable models can be transformed into executable software system implementations in the code generation step of an MDE development process. This enables the automation of software development leading to increased productivity. On the other hand, executable models can be analyzed starting from the early phases of the MDE development process [62]. The analysis facilitates the early validation and verification of the executable models leading to increased quality of the final software system implementation. This is not only of great value in an MDE development process, but also in model-based develop-

ment (MBD), where models are extensively used in the development process, but a complete automation through code generation might not be intended [16, p. 9].

In early phases of the development process, the analysis of executable models aids in an early evaluation of the quality of the software design. Thus, early in the development process, defects in the design can be detected and corrected, which is important as correcting a defect in a system becomes more expensive the later it is detected. Detecting and correcting defects on the model level is especially important in MDE, since the final software system implementation is directly and automatically generated from the models. Besides detecting defects early, executable models enable the evaluation of different design alternatives by evaluating non-functional properties of these alternatives, such as performance and reliability.

Also in the later maintenance phase of the development process, there are important applications of analyzing executable models. In case of required improvements or modifications of the software system due to changed requirements, newly incorporated or modified functionality can be analyzed to evaluate its quality. This new or modified functionality is reflected in the executable models of the software system and, hence, their analysis again aids in the detection of design defects and the evaluation of design alternatives. If an existing system has to be optimized, for instance to handle more client requests concurrently, analyses of the executable models can support identifying optimization potentials of the system, such as performance bottlenecks. Furthermore, alternative optimizations can be evaluated. In case of software migration or software modernization endeavors, analysis aids in detecting parts of a system that are affected and have to be replaced or modified.

Testing, formal analysis, dynamic analysis, debugging, and non-functional property analysis are methods for analyzing executable models. In the following, we discuss these methods to highlight their potential for increasing the quality of software development. For each method, we provide a brief definition, discuss possible applications for executable models, and provide examples of existing approaches applying the respective method on executable models.

2.1.1 Testing

The aim of testing is to validate whether a software system meets its functional requirements, as well as to detect defects in the software system [148]. The former is referred to as *validation testing*, the latter as *defect testing*. Therefore, test cases targeted at the expected use of the software system or at exposing defects are defined. A test case defines input data to the system under test and assertions on the expected output of executing the system on this input. Testing cannot be used to prove that a system is valid and free of defects, rather it is a mean for establishing confidence that the system is sound enough for operational use. Thereby, testing can be applied before each release of the software system, i.e., before it is going into operational use the first time and each time before an updated version is going into operational use. In the latter case, besides testing new or modified functionality of the software system, *regression testing* is an important testing technique. It is concerned with the selective re-testing of the software system to uncover newly introduced defects in previously correctly working functionality.

Testing can be applied to executable models in two ways. Firstly, executable models can be tested themselves. This means that the models are executed on test input data and it is evaluated whether their output corresponds to the expected output. Secondly, executable models can be

used to test the software system implementation, i.e., its source code. In this case, test cases for the implementation are automatically generated from the executable models according to test selection criteria, which ensure the fulfillment of defined test objectives. The obtained test cases are then manually or automatically executed on the software system implementation under test. The executable models used for test case generation might be either the same models as used for code generation, extracted from the software system implementation, or manually created for testing purposes. This is referred to as *model-based testing* [160].

One example for the former case and targeted at executable UML models is the approach proposed by Pilskalns *et al.* [128]. It is concerned with testing UML design models to detect defects in designs early in the development process. Therefore, the design models are first executed symbolically to determine test input data. Using the obtained test input data, the models are then executed and execution traces are recorded, which capture the evolution of the models' states during execution. These execution traces are analyzed to evaluate the assertions on the design models. Based on this testing approach, selective re-testing strategies for regression testing have been proposed [129]. They are concerned with selecting existing test cases, which have to be re-executed for identifying newly introduced defects, and generating additionally needed test cases. Further examples of testing approaches for executable UML models are the testing approaches integrated with the USE tool developed by Gogolla *et al.* at the University of Bremen [55, 56, 59] and the testing approach proposed by Dinh-Trong *et al.* [37–39].

One example of a model-based testing tool operating on executable UML models is the Conformiq Automated Test Design Tools suite[1]. In this tool suite, the external behavior of the system under test is defined by UML state machines. Using symbolic state space exploration algorithms, the UML state machines are symbolically executed and test cases are generated consisting of test input data and test output data. Thereby, different test selection criteria, such as use case coverage, requirements coverage, statement coverage, and all-paths coverage, can be employed for the test case generation. For the generated test cases, executable test scripts for different programming languages, such as Java and C#, can be obtained. Besides UML state machines, also other behavioral diagrams of UML, such as sequence diagrams and activity diagrams, have been employed by existing model-based testing approaches. For surveys on model-based testing approaches and techniques we refer to [36, 69, 160].

2.1.2 Formal Analysis

Formal analysis of software systems aims at proving in a mathematically sound way that a system fulfills certain properties [148]. Therefore, *mathematical models* representing the behavior of a software system are built using *formal specification languages*, such as Z, VDM, B, CSP, finite state machines, process calculi, and Petri nets. These mathematical models are then analyzed using *formal methods*, such as model checking, theorem proving, and SMT solving. Formal analysis is applicable to the validation as well as verification of software systems and especially useful in early stages of the software development process. Unlike testing, formal analysis can prove that a system is valid and free of defects, which constitutes the main advan-

[1] `http://www.conformiq.com/solutions`, accessed 02.06.2014

tage of formal methods over testing. However, formal analysis techniques are more prone to scalability and cost-effectiveness problems than testing approaches.

In the context of MDE, formal analysis can be used to analyze the consistency and correctness of models eventually used for generating the software system implementation, as well as to check certain properties of the models, such as liveness and safety. If models serve as system specification in an MBD process, formal analysis can be used to verify that theses models and, hence, the system specification, are consistent and correct. Furthermore, they can be employed for analyzing whether the manually developed source code of the software system implementation is consistent with its specification defined in terms of models.

In MDE and MBD, formal specification languages and formal methods are usually not directly applied. Instead, the models used in the software development process, such as UML models, are automatically transformed into a mathematical model conforming to a formal specification language. Hence, the developer is not concerned with formal specification languages at all. Furthermore, the formal method is applied automatically and the result is translated back to the original model in order to provide understandable and useful feedback.

Due to the widespread use of UML, several approaches for applying formal analysis to executable UML models exist. One example is the vUML tool developed by Lilius and Paltor [89]. It is a model checking tool for UML models consisting of class diagrams, collaboration diagrams, and state machines. In this tool, the UML models are automatically transformed into PROMELA, which is the input language of the model checker SPIN. SPIN is then used to check dynamic properties of the UML model, such as the absence of deadlocks, livelocks, and reachable invalid state. In case a property is not fulfilled, the counter example generated by SPIN is translated to UML sequence diagrams and provided as result of the analysis. Another example of a formal analysis approach for UML is the work of Eshuis [43]. In this approach, UML models consisting of activity diagrams and class diagrams are transformed into finite state machines in order to check data integrity constraints using the NuSMV model checker.

2.1.3 Dynamic Analysis

Dynamic analysis is concerned with analyzing properties of running programs [6]. Therefore, data about a running program is collected during its execution and analyzed either during or after the execution. The collected data is typically captured in the form of *execution traces*, which provide an abstract representation of the behavior of the executed program. For this purpose, various execution trace formats have been proposed for object-oriented and procedural programming languages [60, 61].

Dynamic analysis can serve different purposes. One of its main purposes is *program comprehension*, that is to support humans in understanding a program's structure and behavior [30]. As pointed out by Lange and Nakamura [84], dynamic analysis is needed for comprehending programs developed with object-oriented languages, because static analysis techniques cannot sufficiently deal with polymorphism and dynamic binding. Therefore, trace visualization and exploration techniques have been developed, which aim at enabling humans to comprehend the large amount of data captured in execution traces [60]. Based on traces, different kinds of metrics can be computed for comprehension purposes. For instance, in the frequency spectrum analysis developed by Ball [6], it is computed how often a program's entities (e.g., operations) have been

executed. Using this information, a program can be decomposed into slices of related computations and its behavior can be related to input and output characteristics. Closely related to comprehension is *debugging*, which is another application purpose of dynamic analysis. In this application, defects in a program are detected by exploring execution traces recored for incorrect executions of a program. Another application of dynamic analysis is *testing*. Unlike traditional testing approaches, which enable the assertion of input/output relations only (cf. Section 2.1.1), dynamic analysis provide the means to also assert expected execution traces. Furthermore, it provides the ability to analyze the test coverage of test suites in order to measure the degree to which the program is tested and identify areas of the program that require more testing. Also *evolution* can be supported by dynamic analysis, as it enables the acquisition of a comprehensive understanding of a program, which is needed for evolving it, as well as of applied evolutions and their impact on the program's behavior. The latter can be supported by the comparison of execution traces captured for different versions of the same program, which makes the effect of the applied evolution explicit on a higher level of abstraction than the differences observable on the source code level. The last application domain we want to mention is *profiling*, which is concerned with measuring the performance of a program.

Executable modeling languages play a crucial role in dynamic analysis, as they are used for visualizing the behavioral aspects captured by execution traces and enable the exploration and analysis of the behavior of a program on a higher level of abstraction. Among these executable languages are, for instance, finite state machines and UML interaction diagrams [30, 60]. The visualization techniques aim at representing the runtime behavior of a system at an appropriate abstraction level, such that humans are able to comprehend and reason about it. In the context of MDE, it is preferable to capture execution traces on the same level of abstractions as the models developed during the system design. Therefore, Maoz [93] proposes the usage of so-called *model-based traces*, which capture the runtime behavior of a system at the same abstraction level as the design models. For instance, in case UML state machines are used during the design, state-based traces can be used, which capture occurred events, performed guard evaluations, and entered and exited states during the program execution [92]. The main advantage of model-based traces is that they filter out irrelevant information about the program execution, such as code statement execution, and add model-specific information not explicitly captured in traditional execution traces and not explicitly visible in the source code, such as the entering and exiting of states. Based on model-based traces, a system's behavior can be analyzed on the same abstraction level as the design models. For instance, coverage metrics can be calculated that are targeted at model elements, such as in the example of state-based traces the number of unique states visited during the execution of the system. Model-based traces have been proposed in the context of models@run.time, a research area within MDE, which is concerned with the development of self-adaptation mechanism of systems based on runtime models [14]. Thereby, runtime models capture the state of a running system and are used to reason about adaptations and apply adaptations during runtime.

When dynamic analysis methods are applied for analyzing programs, their usage is restricted to the later phases of the software development process, where the implementation of the software system is already available. However, dynamic analysis methods can also be applied for analyzing executable models, which enables using them already in early stages of the devel-

opment process for comprehension, debugging, testing, profiling, and evolution purposes. An example of such an application scenario of dynamic analysis methods is performance analysis utilizing simulation techniques, which will be discussed in Section 2.1.5.

2.1.4 Debugging

If defects in the software system have been detected, for instance during the validation and verification process by applying testing, formal analysis, or dynamic analysis methods, these defects have to be located and corrected. For this purpose, debugging is employed [148]. To be able to locate and correct a defect, thorough knowledge about the software system as well as of the defect is necessary. On the one hand, already discussed methods can be used for this purpose. For instance, from the existing test suite, knowledge about the functionality of the system as well as of the defect resulting in a failing test case can be gained. Similarly, a counter example provided by a model checker or an invalid trace revealed by a trace exploration tool can provide the knowledge required to locate and correct the respective defect. On the other hand, specialized methods for debugging exist, which can be utilized for locating defects in a software system.

In code-centric development, *debuggers* provided by the integrated development environment (IDE) used for implementing the software system constitute the most prominent means for debugging. They offer the possibility to *control* the execution of the system, for instance to execute the system in a stepwise manner, i.e., code statement per code statement, pause and resume the execution, set breakpoints for pausing the execution at a specific code statement, and set watchpoints for pausing the execution at accessing a specific variable. Furthermore, debuggers enable the *observation* of the current state of the system by examining the value of existing variables, evaluating expressions on variables, and tracking the current position of the execution. More advanced debugging methods include, for instance, delta debugging and reverse debugging [166].

Besides locating defects in a software system, debuggers can also be used for *testing* purposes and aid in *system comprehension*. For instance, a newly introduced or modified operation can be debugged in order to verify whether it behaves as intended. Similarly, the debugger can be used to explore the behavior of the system by stepwise executing it and observing its state. Thereby, the debugger aids in experiencing and comprehending the system's behavior as well as in experimenting with modifications.

Debugging methods known from code-centric development have also been applied to model-centric development. They are targeted at executable modeling languages and aid in locating defects in models, rapidly testing models, and comprehending the behavior of models already in early stages of the development process. Examples of UML tools providing debuggers for executable UML models are IBM's Rational Software Architect[2] and Rational Rhapsody Developer[3], Sparx Systems' Enterprise Architect[4], NoMagic's Cameo Simulation Toolkit[5] for Magic-

[2] `http://www-03.ibm.com/software/products/en/ratisoftarch`, accessed 10.06.2014

[3] `http://www-03.ibm.com/software/products/en/ratirhap`, accessed 10.06.2014

[4] `http://www.sparxsystems.com/products/ea`, accesses 10.06.2014

[5] `http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`, accessed 10.06.2014

Draw, and the open source tool Pópulo developed by Fuentes *et al.* [51]. These tools provide the capability to control and observe the execution of UML models. For instance, state machines and activities can be executed stepwise, meaning that the execution pauses after the running state machine has processed one event and after one action of a running activity was executed, respectively. Furthermore, breakpoints for model elements, such as states and actions, can be set, such that the execution pauses when these model elements are reached. For enabling the observation of the model execution, the diagrams visualizing the executing models are animated and the values of existing variables are displayed.

2.1.5 Non-Functional Property Analysis

Besides functional requirements, software systems also have to fulfill non-functional requirements. Non-functional requirements are not directly concerned with the concrete behavior of the system, but deal with emergent properties of the system, such as performance, reliability, availability, maintainability, and safety [148]. Considering non-functional properties of a system early in the design process is crucial in order to ensure that the system will meet its non-functional requirements. Thereby, different design alternatives can be analyzed with regard to their impact on the non-functional properties of the overall system in order to identify the best design solution. In this way, costly rework of the software system implementation required due to unfulfilled non-functional requirements can be prevented. This is important, as correcting severe violations of non-functional requirements can require considerable changes in the design, which are more expensive the later they have to be implemented. However, the analysis of non-functional properties may be continued throughout the whole development process of a software system including the maintenance phase, for instance to assess alternative solutions for introducing new components into the system as well as to identify optimization potentials and assess alternative optimizations.

Non-functional properties constitute runtime attributes of a system. Hence, descriptions of a system's runtime behavior are required in order to predict the system's non-functional properties. For this purpose, executable models are employed. As surveyed by Balsamo *et al.* [7], in the realm of performance prediction, the most used executable modeling languages are queuing networks, stochastic Petri nets, stochastic process algebras, and simulation models. These models are analyzed using either analytical methods or simulation techniques in order to obtain performance indices, such as response time, utilization, and throughput. For analyzing the dependability of a software system, consisting of reliability, availability, maintainability, integrity, and safety, executable modeling languages, such as fault trees, Markov chains, and Petri nets are employed [11]. Again through the application of analytical methods or simulation techniques, dependability indices, such as mean time to failure, failure rate, mean time to repair, steady state availability, and the safety integrity level are computed.

Because UML is today the most adopted modeling language for designing software systems, considerable efforts have been undertaken to integrate existing techniques for analyzing non-functional properties with UML [7, 11]. An important contribution in this direction was made by the standardization of the UML profile MARTE [115], which enables the extension of UML models with information required for performance and schedulability analysis. For instance, information about resources, scenarios, and workloads can be added to UML models using

MARTE. Based on a UML model defining the structure and the behavior of a software system and extended with additional information about factors influencing the non-functional properties of the software system, the non-functional properties can be quantitatively analyzed. Therefore, two different approaches are used. In the first approach, the UML model is transformed into the modeling language used by the respective analysis technique, such as queuing networks or Petri nets. In the second approach, the analysis technique is directly implemented for UML.

In performance engineering, dedicated interchange formats have been elaborated to reduce the effort needed for transforming UML models into performance analysis models. The Core Scenario Model (CSM) proposed by Petriu and Woodside [127] constitutes such an interchange format. The PUMA framework of Woodside *et al.* [164] proposes an architecture of a performance analysis tool chain, where software design models are first transformed into CSM models, which are then again transformed into models suitable for different kinds of performance analysis. Woodside *et al.* also introduce algorithms for transforming UML models extended with applications of the UML Profile for Schedulability, Performance, and Time (SPT) [110] (the predecessor of MARTE) to CSM models, as well as algorithms for transforming CSM models into queuing networks, layered queuing networks, and Petri nets. The obtained models serve as input to performance analysis tools to compute performance indices predicting the performance of the modeled software system. Another interchange format for performance analysis is the Performance Model Interchange Format (PMIF) developed by Smith *et al.* [144].

Balsamo *et al.* [7] surveyed approaches for integrating model-based performance prediction into the software development process. Thereby, out of 15 investigated methodologies for performance analysis, nine are based on UML models. In the area of dependability analysis, Bernardi *et al.* [11] surveyed approaches for modeling and analyzing dependability properties of software systems. Thereby, the survey deliberately investigates approaches explicitly targeted at UML models. Both approaches based on the transformation of UML models into analysis models as well as approaches directly analyzing UML models have been identified and investigated by this survey.

Methods and techniques for performing non-functional property analysis are important for increasing the quality of software systems. By integrating these methods and techniques with executable modeling languages used in MDE, the fulfillment of non-functional requirements may be verified from the early phases of the development process on.

2.2 Defining Modeling Languages

In the previous section, we emphasized the potential of analysis methods targeted at executable models for increasing the quality of the final software system. However, in order to enable the utilization of models—being it for code generation or analysis purposes—the used modeling languages have to be formally defined, such that conforming models can be processed by computers. The formal definition of modeling languages is the subject of this section.

As depicted in Figure 2.1, the definition of a modeling language consists of the definition of the language's concepts and notations referred to as *abstract syntax* and *concrete syntax*, respectively, as well as the definition of the language's meaning referred to as *semantics* [5, 16, 75, 156, 162].

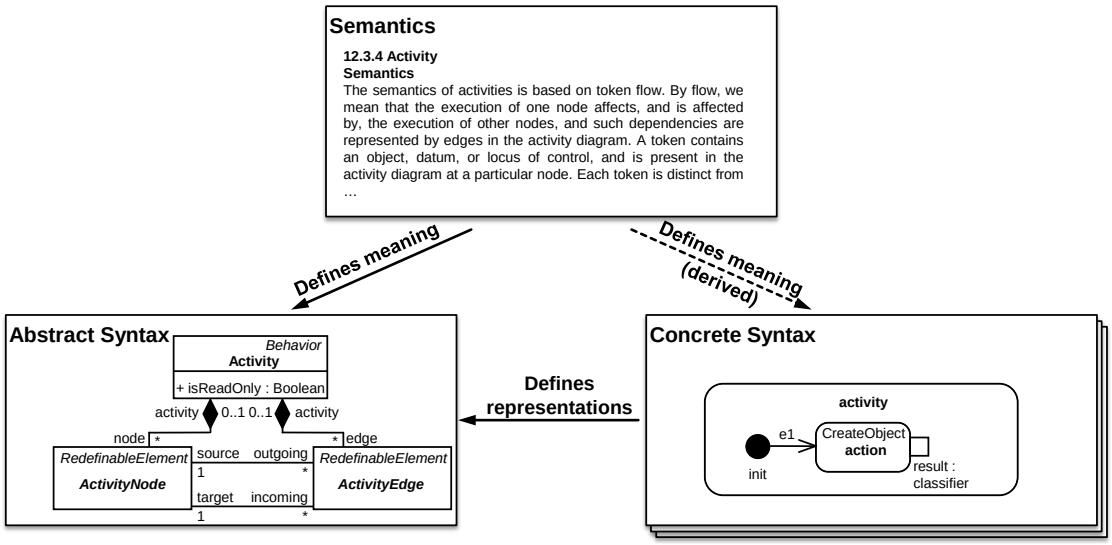


Figure 2.1: Components of a modeling language definition (adopted from [16])

The *abstract syntax* of a modeling language defines the *modeling concepts* that can be used to construct models for describing a real world phenomena on a suitable level of abstraction. Each modeling concept consists of attributes and relations to other modeling concepts. Therewith, the abstract syntax defines the structure of models that can be constructed with the respective modeling language. The standard means for formally defining the abstract syntax of a modeling language are *metamodels*. Figure 2.1 depicts a very small excerpt of UML's metamodel, which defines the modeling concepts Activity, ActivityNode, and ActivityEdge, as well as the relations between these modeling concepts. In Section 2.2.1, we discuss the technique of metamodeling for defining the abstract syntax of a modeling language in more detail.

The *concrete syntax* of a modeling language defines the *representation of models* conforming to this modeling language. It constitutes the interface of a modeling language that is usually used by modelers to construct models. Multiple concrete syntaxes can be defined for a modeling language. Generally, it can be distinguished between *textual* and *graphical* concrete syntaxes, defining textual and graphical representations for the modeling concepts provided by the modeling language, respectively. In the case of graphical concrete syntaxes, a model is usually represented by a set of *diagrams*, each representing a certain part of the model or a certain view on the model. Figure 2.1 depicts the graphical concrete syntax of UML for representing the modeling concepts Activity, ActivityNode, and ActivityEdge in activity diagrams.

The *semantics* of a modeling language defines the *meaning* of models constructed with this modeling language [63]. Therefore, the abstract syntax of the modeling language is mapped to a semantic domain via a semantic mapping. The *semantic domain* is a well defined and well understood domain suitable to express the meaning of models conforming to a specific modeling language. The *semantic mapping* maps the syntactic elements of the modeling language, i.e., the provided modeling concepts, to elements of the semantic domain. Ideally, the semantic domain

and the semantic mapping are defined formally, such that an automated semantic manipulation and analysis of models is possible. However, they can be defined in various degrees of formality, from descriptions in natural language to rigorous mathematical definitions. For instance, the semantic domain of UML is informally defined in English prose. Figure 2.1 depicts an excerpt of the description of the semantics of the UML modeling concept Activity. For defining the semantics of modeling language formally no standard means have been established yet. In Section 2.2.2, we discuss ways of formally defining the semantics of a modeling language, which are proposed by existing work.

2.2.1 Metamodeling

Metamodels constitute the standard means for formally defining the abstract syntax of modeling languages. Thereby, the *metamodel* of a modeling language defines the language's modeling concepts, which can be used to construct conforming *models*. For defining metamodels, metamodeling languages are used, which are themselves defined by *meta-metamodels*. The dependencies between models, metamodels, and meta-metamodels result in a three-layered language definition hierarchy referred to as *metamodeling stack* [16, 81]. This metamodeling stack is conceptually depicted in Figure 2.2. A model represents some real world phenomena as part of a system and is located on the layer M1 of the metamodeling stack. It is expressed using the modeling concepts provided by a modeling language, which are defined by the metamodel of the modeling language located on layer M2. Therewith, it is common to regard the model as an instance of the metamodel defining the used modeling language. The metamodel of a modeling language is itself expressed using a metamodeling language, which is defined by a meta-metamodel located on layer M3. Thus, just as the model may be regarded as an instance of a metamodel, a metamodel may be regarded as an instance of a meta-metamodel. However, please note that strictly speaking a model located on one layer M_X is not an instance of a model located on the next higher layer M_{X+1} in the sense of object-class instantiation relationships known from object-oriented programming, but instead the model on M_X has to conform to the model on M_{X+1} [12,138]. Thus, the *instance of* relationship between models located on different layers of the metamodeling stack is also referred to as *conforms to* relationship. It may not be confused with the *instance of* relationship known from object-oriented programming to denote the relationships between objects and classes.

In the following, we explain the metamodeling stack in detail starting with the layer M3. In this explanation, we make use of the exemplary instantiation of the metamodeling stack depicted in Figure 2.3, which employs the metamodeling language Ecore for defining selected modeling concepts of UML that may be used to model a system.

M3 meta-metamodel. A meta-metamodel defines a metamodeling language, which is used to define metamodels (cf. Figure 2.2). Therefore, a meta-metamodel defines a set of metamodeling concepts that can be instantiated for constructing metamodels. Meta-metamodels are defined reflexively, meaning that a meta-metamodel can be defined using the very same meta-metamodel and, hence, can be regarded as an instance of itself.

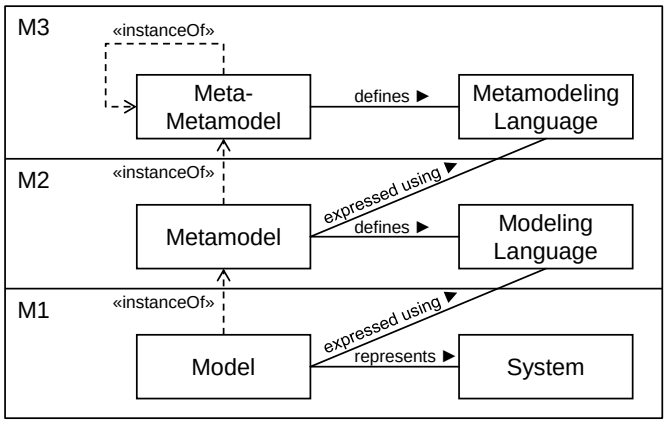


Figure 2.2: Metamodeling stack (adopted from [16, 81])

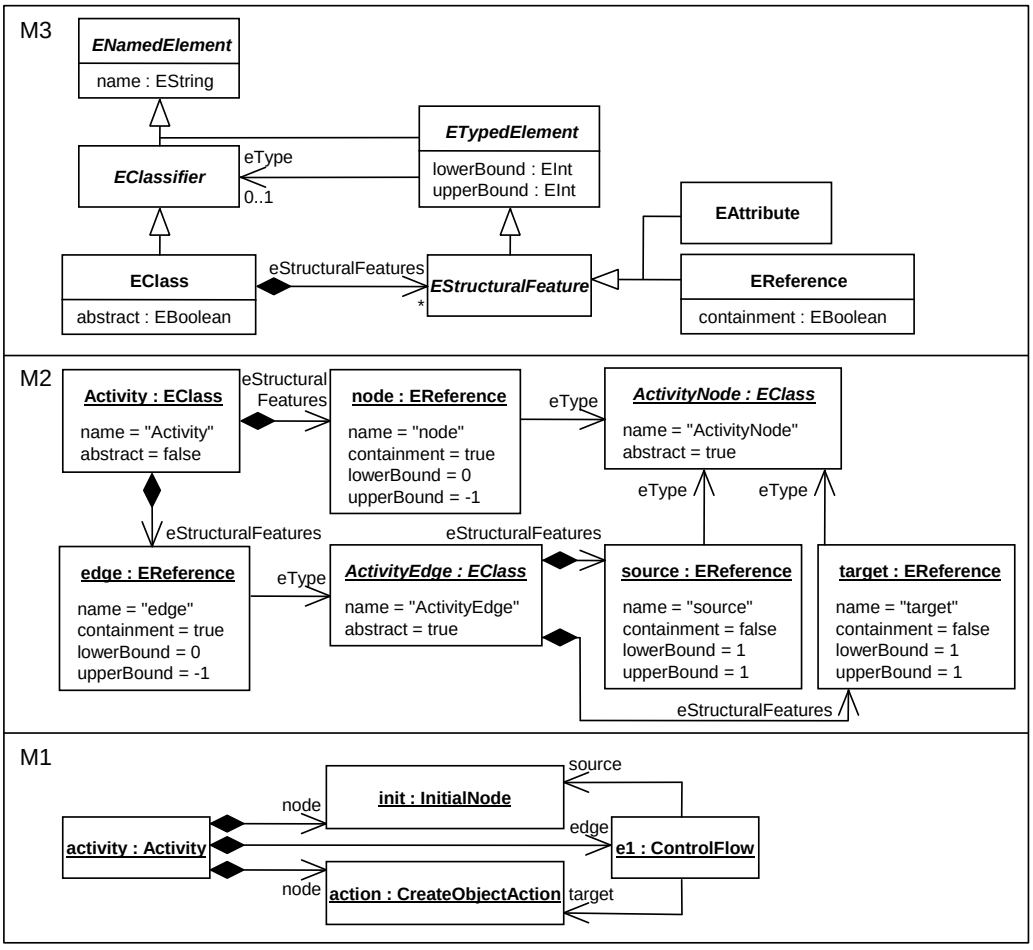


Figure 2.3: Metamodeling stack example: Ecore

In 1997, OMG adopted the meta-metamodel of the metamodeling language MOF [121] to ensure the interoperability of metamodels and modeling environments [13]. The metamodels of all modeling languages standardized by OMG, such as UML, SPEM, and CWM, are defined using MOF. MOF is defined reflexively, meaning that it is defined in itself. Since version 2.0 of the MOF standard, the meta-metamodel of MOF is a subset of the metamodel of UML, which comprises UML modeling concepts for defining classes consisting mainly of attributes and associations to other classes. This relation between MOF and UML clearly shows the possibility to reflexively define MOF with MOF itself, as well as to reflexively define UML with UML itself.

Another prominent metamodeling language is *Ecore*, which is the metamodeling language of the open source modeling environment EMF [150]. EMF is built on top of the Eclipse platform and provides tool support for MDE, which is widely used in academia and practice. Based on the definition of a metamodel with Ecore, EMF provides facilities for persisting conforming models, generators for tree-based modeling editors and APIs for programmatically accessing and manipulating models, as well as an reflection API. Around EMF, a whole ecosystem of tools supporting MDE emerged comprising tools for abstract syntax development, concrete syntax development, model development, model transformation, and code generation[6].

Figure 2.3 shows an excerpt of Ecore's meta-metamodel defining the metamodeling concepts EClass, EAttribute, and EReference. The metamodeling concept EClass can be used to define the concepts of a modeling language in a metamodel referred to as *metaclasses*. Thereby, the definition of a metaclass consists of a name (attribute name inherited from ENamedElement), the information whether the metaclass can be directly instantiated in a model or not (attribute abstract), and of a set of structural features (reference eStructuralFeatures). A structural feature is either an attribute (metamodeling concept EAttribute) or a reference to another metaclass (metamodeling concept EReference). Structural features have a name (attribute name inherited from ENamedElement), a multiplicity (attributes lowerBound and upperBound inherited from ETypedElement), as well as a type (reference eType inherited from ETypedElement). Attributes can be of one of the primitive data types predefined by Ecore, such as EString, EBoolean, and EInteger for representing String, Boolean, or Integer attributes. References refer to metaclasses as type and either constitute cross references or containment references (attribute containment).

M2 metamodel. A metamodel is an instance of a meta-metamodel and defines a modeling language (cf. Figure 2.2). More precisely, a metamodel defines the abstract syntax of a modeling language consisting of the modeling concepts that can be used to construct models with this language. Thereby, each element of a metamodel is an instance of an element of the used meta-metamodel.

Figure 2.3 shows an excerpt of the UML metamodel, which defines the modeling concepts Activity, ActivityNode, and ActivityEdge as well as the relations between them. Therefore, the metamodel consists of three instances of Ecore's metamodeling concept EClass with the names *"Activity"*, *"ActivityNode"*, and *"ActivityEdge"*. Furthermore, it contains four instances of EReference named *"nodes"*, *"edges"*, *"source"*, and *"target"*. The former two references define that an activity contains arbitrarily many activity nodes as well as arbitrarily many activity edges. The latter two references define that an activity edge refers to exactly one source activity

[6] `http://www.eclipse.org/modeling`, accessed 23.06.2014

node and to exactly one target activity node. Please note that Figure 2.3 depicts the UML metamodel excerpt in object diagram notation to explicitly show the *instanceOf* relations between the elements located on the layers M2 and M3.

M1 model. Finally, a model is an instance of a metamodel representing some real world phenomena as part of a system (cf. Figure 2.2). Thus, each element of a model constitutes an instance of an element of the metamodel of the used modeling language.

Figure 2.3 depicts an excerpt of a UML model consisting of an activity containing two activity nodes and one activity edge connection them. Therefore, the model consists of one instance of the metaclass Activity, two instance of the metaclass ActivityNode (in fact of two subclasses of ActivityNode, namely InitialNode and CreateObjectAction), one instance of the metaclass ActivityEdge (in fact of the subclass ControlFlow), as well as references between these instances. Again we use the object diagram notation for representing the model.

The common acceptance of metamodels for defining the abstract syntax of modeling languages, as well as the standardization of the metamodeling language MOF, paved the way for the emergence of a variety of techniques for automating the development of tools supporting the syntactic manipulation and analysis of models. In particular, it provided the basis for developing *generic* techniques for processing models independent of the used modeling language. These techniques process models based on the metamodel they conform to by leveraging only knowledge about the underlying metamodeling language, but not about the specific metamodel itself. For instance, model transformation and code generation engines are capable of processing any kind of model as long as it conforms to a metamodel defined with a known metamodeling language. Furthermore, *generative* techniques for automating the tool development have been elaborated. Examples are techniques for automatically generating modeling editors from metamodels.

2.2.2 Semantics Specification

In the field of programming language design, efforts targeted at providing means for formally defining semantics have a long history starting in the late 1960s. Thereby, it is distinguished between three general approaches, namely denotational semantics, operational semantics, and axiomatic semantics [108]. In the *denotational semantics* approach, a program's meaning is defined abstractly as a mathematical object representing the effect of executing the program. In contrast, in the *operational semantics* approach, a program's meaning is defined in terms of the steps of computation performed to execute the program. In the third general approach, that is the *axiomatic semantics* approach, a program's meaning is defined as a set of axioms and rules satisfied by the program's execution.

In MDE, two distinct approaches for formally defining the semantics of modeling languages have been applied, namely the *translational semantics* approach and the *operational semantics* approach [28,77]. While these two approaches have commonalities with the denotational semantics approach and the operational semantics approach known in the field of programming language design, most of the existing translational and operational semantics approaches in MDE do not rest upon an equally rigorous mathematical fundament. Instead, existing translational

and operational semantics approaches propose the implementation of *compilers* and *interpreters* for modeling languages, respectively, as known from the implementation of programming languages.

In the following, we provide a general description of the translational and operational semantics approach applied in MDE, present concrete examples of these approaches, and discuss their advantages and disadvantages.

Translational semantics. In the *translational semantics* approach, the semantics of a modeling language is defined by a *translation* from the modeling language to another language whose semantics is already formally defined. The modeling language whose semantics is to be defined is referred to as *source language* and the language to which the source language is translated is referred to as *target language*. In the translation, the concepts provided by the source language are translated into concepts provided by the target language. Thereby, the translation of one instance of a concept provided by the source language may result in several instances of distinct concepts provided by the target language. For realizing this translation, model transformation techniques may be employed. In the translational semantics approach, the target language constitutes the semantic domain of the (source) modeling language's semantics definition. The translation from the source language to the target language constitutes the semantic mapping of the semantics definition.

An example of the application of the translational semantics approach is the definition of the semantics of UML activities developed by Störrle [151–155]. In this work, the semantics of UML activities is formally defined by a translation to Petri nets. In this translation the modeling concepts provided by UML for defining activities are translated into transitions, places, and arcs of Petri nets. For instance, an action is translated into a transition, a decision nodes is translated into a place, and an activity edge is translated into a place with one incoming and one outgoing arc [154].

One special kind of translational semantics approach is the approach proposed by Chen *et al.* called *semantic anchoring* [23]. In this approach, the semantics of a (source) modeling language is defined by a translation from the modeling language to a so-called *semantic unit*, which is defined with another (target) language whose semantics is formally defined. Thereby, the definition of a semantic unit comprises the definition of the syntax as well as the semantics of the semantic unit. The semantics of the semantic unit is usually defined using the operational semantics approach discussed later in this section. Through the translation between the modeling language and the semantic unit, the semantics of the modeling language is *anchored* to the semantic domain of the semantic unit. Chen *et al.* developed an infrastructure enabling the application of the semantic anchoring approach, which makes use of the abstract state machine language as target language to define semantic units. Using this infrastructure, they defined the semantics of several modeling languages, such as finite state machines.

The advantage of the translational semantics approach is that tools available for the used target language can be reused for the source language. Thus, tools supporting the analysis of models conforming to the target language may be used for analyzing models conforming to the source language. The drawback of this approach, however, is that the semantics of a modeling language is defined by the translation to the target language leading to an additional level of

indirection. This additional level of indirection affects the utility provided by the reused tools available for the used target language, as results obtained from these tools are only available in the target language. For making actual use of them, they have to be translated back from the target language to the source language. Defining both kinds of translations—the translation from the source language to the target language as well as translations of results from the target language to the source language—is a complex task due to the deep knowledge required not only about the source language but also about the target language.

Operational semantics. In the *operational semantics* approach, the semantics of a modeling language is defined by specifying the steps of computation required for executing a model conforming to the modeling language. Thus, the operational semantics defines an *interpreter* for the modeling language, which can be regarded as *state transition system* defining how an executing model progresses from state to state. Therefore, the operational semantics defines on the one hand *runtime concepts* needed for capturing the state of an executing model constituting the semantic domain of the semantics definition and on the other hand the *steps of computation* involved in performing transitions of the executing model from one state to another state constituting the semantic mapping of the semantics definition. While the runtime concepts needed for defining the state of an executing model can be defined by applying metamodeling techniques, the steps of computation progressing the executing model to a new state has to be defined with an executable language. Executable languages usable for defining operational semantics include programming languages, action languages, and model transformation languages.

One example of a *programming language* usable for defining operational semantics is Java in combination with the Java API of EMF [150], which allows the access to and manipulation of models conforming to a modeling language defined with Ecore. While programming languages provide much expressive power, their usage for defining operational semantics has the following drawbacks. It requires language designers to have high programming skills, forces the language designer to leave the metamodeling environment, compromises the platform independence of the modeling language, and impedes the analysis of operational semantics. Due to these drawbacks, action languages and model transformation languages are preferable over programming languages.

An *action language* usable for defining operational semantics has to be integrated with metamodeling languages, such that it enables the access to and manipulation of models. One example of an action language proposed to define operational semantics is Kermeta [72, 107]. It is an imperative, object-oriented, and aspect-oriented action language integrated with Ecore. With Kermeta, the operational semantics of a modeling language is defined by weaving aspects into the modeling language's metamodel, which extend the defined metaclasses with additional attributes, references, as well as operations and which can also introduce new metaclasses into the metamodel. By extending existing metaclasses with additional attributes and referenced as well as by introducing additional metaclasses, runtime concepts for capturing the state of an executing model are defined. Introduced operations define the steps of computation involved in the execution of a model. For defining these steps of computations, Kermeta provides on the one hand usual imperative statements, such as block statements, loop statements, conditional statements, and assignment statements, and on the other hand OCL-like expressions. Other action

languages proposed to be used for defining operational semantics include XOCL [24], which extends OCL with actions enabling to express manipulations of models, and the action language developed by Scheidgen and Fischer [136], which is based on UML activities and OCL.

The third alternative type of languages usable for defining operational semantics are *model transformation languages*. As an example, we want to briefly introduce the approach proposed by Engels *et al.* called *dynamic meta modeling (DMM)* [41, 64, 145], which makes use of graph transformations to define operational semantics. In DMM, the runtime concepts defining the state of an executing model are defined by means of an own metamodel called *runtime metamodel*. Transitions of an executing model between states are defined by means of operational graph transformation rules using DMM's own graph transformation language. The abstract syntax of this graph transformation language is defined by means of a metamodel and its concrete syntax is inspired by UML communication diagrams allowing the definition of graph transformation rules in a graphical way. The semantics of this language is defined in terms of a translation to the graph transformation tool set GROOVE [132]. Thereby, the runtime metamodel defined for a modeling language is translated into a type graph and the operational DMM graph transformation rules are translated into GROOVE graph transformation rules. For executing a model, it is translated into a GROOVE host graph being an instance of the obtained GROOVE type graph and the obtained GROOVE graph transformation rules are applied to this graph. Other operational semantics approaches relying on model transformation languages are the approaches by Kastenberg *et al.* [74] proposing the direct use of GROOVE graph transformation rules for operationally defining the semantics of object-oriented and imperative languages, Rivera and Vallecillo [134] using Maude rewriting rules [26], Rivera *et al.* [133] using ATOM3 graph transformation rules [34], and Sadilek and Wachsmuth [135, 163] using QVT Relations [112].

The advantage of the operational semantics approach compared to the translation semantics approach is, that the semantics is defined directly for the considered modeling language instead of indirectly through the translation to another language. However, the drawback of this approach is that semantics-based tools, such as tools supporting the analysis of models, have to be newly developed for each modeling language.

Although several languages and techniques have been developed and applied for formally defining the semantics of modeling languages, none of them is commonly accepted in the MDE community. This results on the one hand in modeling languages having no formally defined semantics and on the other hand in the need for manually implementing semantics-based tools supporting, for instance, in the analysis of models. We believe that a standardized, well established, and widely accepted semantics specification language constitutes a crucial prerequisite for establishing formal semantics definitions as integral part of modeling languages and paves the way for the emergence of methods and techniques automating the development of semantics-based tools.

2.3 Executable UML

In the previous section, we discussed the need for formally defining modeling languages in order to enable the automated processing of conforming models. Thereby, we particularly highlighted

the potential of automating the development of tool support for modeling languages based on their formal definition. The formal definition of a modeling language's abstract syntax in terms of metamodels enables the automation of the development of syntax-based tools, such as modeling editors. However, the lack of a commonly accepted way for formally defining the semantics of a modeling language impedes the automated development of semantics-based tools, such as model debuggers, model testing environments, and dynamic model analysis tools. Formally defining the semantics of a modeling language—being it in a translational or operational way—requires a language serving as the semantic domain, whose semantics is already formally defined. In this thesis, the usage of a subset of UML for formally defining the behavioral semantics of executable modeling languages is investigated. The semantics of the investigated subset was formally defined and standardized by OMG in the fUML standard [114]. This standard was only adopted in 2008, eleven years after the adoption of the UML standard itself in 1997. In this section, we provide a summary of fUML's history, present an overview of other existing formalizations of UML's semantics, and give a brief introduction to fUML.

In 1995, the history of UML began, when Grady Booch and James Rumbaugh published the *Unified Method Version 0.8* being a unification of their object-oriented development methods called the *Booch-Method* and the *Object Modeling Technique (OMT)*. One year later, Ivar Jacobsen, who proposed the *Object-Oriented Software Engineering approach (OOSE)*, joined the effort to provide a *Unified Modeling Language (UML)* for object-oriented development. The original intent of the standardization of UML was primarily the *unification* of the various existing graphical modeling languages supporting object-oriented analysis. Another important aim of the standardization was to enable the syntactic *interchange* of models between different modeling tools. In 1997, UML 1.1 was officially adopted by OMG and quickly became the de facto standard for visualizing, specifying, constructing, and documenting the artifacts of software systems.

Until version 1.5 of UML adopted in 2002, UML was *not executable* because it provided only a very limited set of actions for expressing the behavior of a software system [101]. With UML 1.5, however, the *action semantics* was integrated with UML. The aim of the action semantics was to evolve UML to a computationally complete language and therewith support model-based simulation and verification of system specifications, as well as full code generation [102, 109]. Therefore, a set of primitive actions was defined enabling the precise and complete specification of computations to be carried out by a software system in a software-platform-independent way.

One of the main advocates of the action semantics was Stephen Mellor, well known for the Shlaer-Mellor method [143] at this time. Based on the action semantics, he proposed the executable UML development method *xUML* aiming at providing the means for defining the behavior of software systems with executable models detailed enough to compile them into source code [101]. For expressing the behavior with xUML, UML state machines and the UML actions defined by the action semantics were used. For the latter, Mellor *et al.* defined their own action language providing a textual syntax.

Despite the introduction of the action semantics in UML 1.5 and its persistence in the 2.x versions of UML, the specification of the semantics of actions and of UML in general remained imprecise and incomplete. In the UML specification, the semantics of the provided modeling

concepts is defined informally in English prose. Moreover, up to and including versions 2.4.x of UML, this informal definition is scattered throughout the specification. This leads to several problems, as pointed out repeatedly (cf. for instance [18, 50, 66]). One of these problems is that it is difficult to gain a global understanding of UML's semantics because the semantics definition is much dispersed in the specification. As a result of this, there are logical inconsistencies and omissions in the semantics definition. However, with the latest version of UML, that is version 2.5 [118], the specification of UML has been substantially simplified and improved in order to eliminate redundancies and inconsistencies, as well as to aid in understanding the semantics of the modeling concepts provided by UML. Nevertheless, having the semantics of UML only informally defined may lead to ambiguities and different interpretations of UML. This can lead to serious problems in the software development process, to difficulties in learning and using UML, as well as to incompatibilities of UML tools. As a result of this criticism, much research effort has been devoted to formalizing the semantics of UML. These efforts have resulted in various formalizations of the semantics of subsets of UML. We will provide an overview of these endeavors in Section 2.3.1.

However, the emerged formalizations of UML's semantics are targeted at different versions and different subsets of UML. Furthermore, distinct semantics specification approaches and semantic domains are used. In response to the need for a standardization of a precise semantics of UML, OMG issued the request for proposal for the *Semantics of a Foundational Subset for Executable UML Models* in April 2005. Three years later, in 2008, the resulting foundational UML (fUML) standard was adopted by OMG, and finally published in February 2011 [114]. The fUML standard provides a formal specification of the semantics of the foundational core of UML comprising a subset of UML's class modeling concepts, activity modeling concepts, and action language. In Section 2.3.2, we provide a brief introduction to fUML.

2.3.1 Formalization of UML's Semantics

A huge body of work concerned with the formalization of UML's semantics exists in the literature. Thereby, the individual formalizations address specific *subsets of UML*, utilize specific *semantic domains*, and target specific *application purposes*.

In his thesis, Hausmann [64, pp. 23–28] surveys 25 distinct formalizations of UML's semantics. The surveyed formalizations address specific subsets of UML, which include modeling concepts of class diagrams, use case diagrams, activity diagrams, interaction diagrams, and state machines. Furthermore, specific semantic domains are utilized for the formalization, such as the process specification language, abstract state machines, symbolic transition systems, CSP, and Petri nets. The purposes of the formalizations include the provision of a formal semantic specification, consistency checking, and verification. A similar survey was undertaken by O'Keefe [124], who discusses several formalizations of UML class diagrams, state machines, interaction diagrams, and OCL utilizing different semantic domains, such as symbolic transition systems, CSP, Z, graph transformations, and higher order logic.

Crane and Dingle [31] categorized and compared 26 different formalizations of UML state machines. They point out that these formalizations vary widely regarding the coverage of UML modeling concepts for defining state machines. The identified semantic domains utilized by the surveyed formalizations include abstract state machines, transition systems, Petri nets, rewrit-

Formalization	UML version	Actions	Approach	Semantic domain	Application
Börger *et al.* [15]	1.3	No	Translational	Abstract state machines	Formalization
Crane and Dingel [32, 33]	2.1	Yes	Translational	System model (mathematical model)	Formalization, dynamic analysis
Engels *et al.* [42]	2.0	No	Operational	Graph transformations	Formalization, formal analysis
Eshuis and Wieringa [43–45]	2.0	No	Translational	Transition systems	Formalization, formal analysis
Grönniger *et al.* [58]	2.2	No	Translational	System model (mathematical model)	Formalization
Störrle *et al.* [151–155]	2.0	Partly (communication)	Translational	Petri nets	Formalization, dynamic analysis, formal analysis

Table 2.1: Overview of formalizations of the semantics of UML activities

ing systems, model checking languages, and other formal specification languages. Similarly, Micskei and Waeselynck [103] surveyed 13 formalizations of UML sequence diagrams for investigating the addressed UML modeling concepts, utilized semantic domains, and differences in the defined semantics.

Because this thesis is concerned with utilizing fUML and in particular its modeling concepts for defining activities, we provide a brief overview of work concerned with the formalization of the semantics of UML activities. Table 2.1 provides a summary of six distinct formalizations of the semantics of UML activities including information about which version of UML is considered, whether the semantics of UML's action language is formalized, which semantics specification approach is applied, which semantic domain is utilized for the formalization, as well as which application purpose of the formalization is discussed in the respective work. While all formalizations except one consider version 2 of UML, only the formalization provided by Crane and Dingel [32, 33] takes a considerable subset of the actions provided by UML into account [32, p. 684]. Five of the formalizations apply a translational approach utilizing abstract state machines, the mathematical model called system model, Petri nets, and transition systems. An operational approach is applied by Engels *et al.* [42] utilizing graph transformations. Regarding the purpose of the formalization some approaches only aim at providing a formal semantics of UML activities, while others show the applicability of the respective formalization for the purpose of dynamic analysis and formal analysis. Due to the consideration of UML's action language, the work of Crane and Dingel [32, 33] can be considered as formalization most related to the fUML standard.

This overview of existing formalizations of UML's semantics should make clear that there is a high demand for a standardized semantics of UML in the MDE community and, thus, should emphasize the importance of the fUML standard.

2.3.2 Foundational UML (fUML)

The fUML standard [114] selects a foundational core of UML, the so-called *foundational UML subset* or *fUML* in short, and provides a precise specification of its behavioral semantics. The fUML subset is a strict subset of UML 2.3 [111] meaning that it does not modify the abstract syntax of the selected subset. However, the fUML standard defines additional well-formedness rules for this strict subset in terms of OCL constraints [120]. For modeling static aspects of a system, i.e., a system's structure, the fUML subset contains modeling concepts for defining UML classes. For modeling dynamic aspects of a system, i.e., a system's behavior, the fUML subset contains modeling concepts for defining UML activities. The modeling concepts for defining UML activities comprise a subset of UML's action language consisting of predefined actions for expressing object manipulations and communications between activities. Besides actions, control nodes for expressing control flows and object nodes for expressing data flows are included in fUML. As stated before, version 1.0 of fUML was published in February 2011 [114] and selects a subset of UML 2.3 [111]. However, the current version of fUML at the time of writing this thesis is version 1.1 [119], which selects a subset of UML 2.4.1 [113]. In version 1.1 minor defects in the fUML standard have been corrected and the primitive data type Real, as well as the flow final node, which is a control node, have been added to the fUML subset.

The behavioral semantics of the fUML subset is specified precisely by the *fUML execution model* in an operational way. Therefore, an even smaller subset of fUML, called the *base UML (bUML) subset*, is used to define a virtual machine for executing fUML models. In the fUML standard, Java is used as a surface notation for defining the fUML execution model. The semantics of bUML is in turn defined with the first-order logic formalism *Process Specification Language (PSL)* [71] following a translational semantics approach.

The fUML subset is considered as the foundation of the UML modeling language on which the remainder of the language resides. Hence, fUML is intended as being sufficient for specifying the semantics of the entire UML modeling language, i.e., not only the UML modeling concepts contained by the fUML subset, but also all not contained UML modeling concepts. This is due to the fact, that it is possible to translate these modeling concepts to the fUML subset [114, p. 19]. However, such a translation is not standardized yet and out of scope of the fUML standard.

In this respect, efforts to standardize the semantics of UML composite structures are currently undertaken by OMG. The corresponding standard is called *Precise Semantics of UML Composite Structures (PSCS)* and has at the time of writing the status of an OMG adopted beta specification [122], meaning that it is adopted by OMG and currently in the finalization phase. UML composite structures are classifiers that have an internal structure, meaning that they consist of parts. Structured classifiers are connect with each other and with their internal parts via ports and connectors. The semantics of UML composite structures comprises on the one hand the runtime manifestation of parts, ports, and connectors during model execution, and on the other hand the life-cycle of composite objects and their parts as well as the behavior of flows

through ports and connectors. For specifying this semantics, an approach different from the translational approach foreseen in the fUML standard and mentioned earlier was chosen. Instead of translating the additional UML modeling concepts for defining composite structures to fUML, the fUML subset as well as the fUML execution model have been extended.

Another standard complementary to the fUML standard is the *Action Language for Foundational UML (Alf)* standard [117] adopted in 2010 and finalized in 2012. While the UML standard provides a graphical concrete syntax for its modeling concepts, which is adopted by the fUML standard for representing the UML modeling concepts contained by the fUML subset, the Alf standard provides a textual concrete syntax for the fUML subset. Using Alf, both structure comprising classes and behavior comprising activities can be defined completely textually. However, as explicitly stated by the Alf standard [117, p. 1] it is also possible to combine the graphical and textual notation, for instance, by only defining activities textually. The semantics of Alf is defined in the standard by a mapping to fUML. However, tool vendors can alternatively choose to directly interpret Alf code or translate it to some other executable form.

In Chapter 3, we provide a thorough overview of the fUML standard comprising a description of the UML modeling concepts contained by the fUML subset, as well as the fUML execution model defining the fUML virtual machine capable of executing fUML models.

CHAPTER 3

Foundational UML

3.1 Introduction

As introduced in Section 2.3.2, the aim of the fUML standard [114] is to identify the *foundational core of UML* and to provide a precise and complete specification of the *behavioral semantics* of this foundational core. Thereby, the foundational core of UML is considered as the fundament of the UML modeling language and its behavioral semantics constitutes the foundation for eventually defining the behavioral semantics of the remainder of UML. Before we dive into the details, we provide an overview of fUML's language definition. Figure 3.1 depicts the components of fUML's language definition and points to the associated standard.

Abstract syntax. The fUML standard selects a foundational core of UML comprising modeling concepts for defining the structure of a system with classes and the behavior of a system with activities. This foundational core is referred to as *fUML subset*. Therefore, a strict subset of UML's metamodel defined in the UML standard [113] is selected, which constitutes the abstract syntax definition of fUML. Furthermore, the fUML standard defines additional well-formedness rules for certain modeling concepts using OCL, which are necessary to precisely define the semantics of the respective modeling concept.

Concrete syntax. For representing fUML models, three alternatives are available. The first alternative is to use the *graphical notation* defined by the UML standard [113] for the modeling concepts contained by the fUML subset, that is the class diagram notation as well as the activity diagram notation. Alternatively, fUML models can be represented in the *textual concrete syntax* defined by the Alf standard [117]. As third alternative, a *mixture* of the graphical representation and the textual representation can be used. For instance, classes could be defined using the graphical class diagram representation, whereas activities defining the behavior of class operations could be represented textually in Alf syntax.

In this thesis, the graphical notation of class diagrams and activity diagrams defined by the UML standard is used.

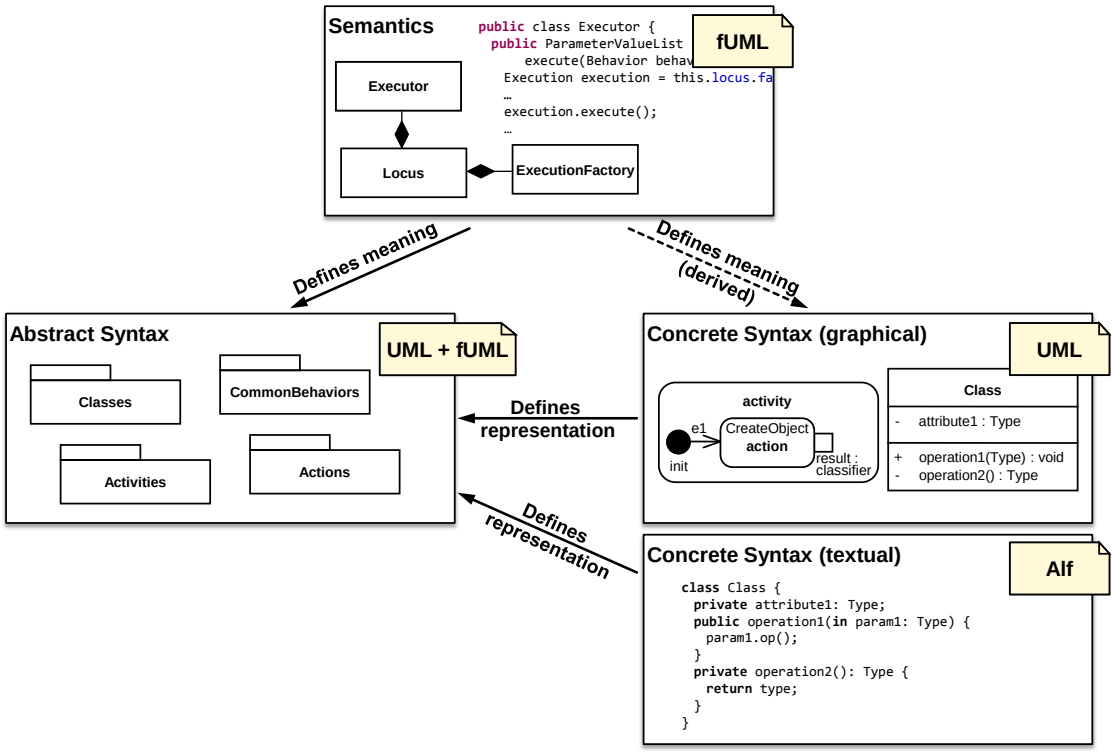


Figure 3.1: Components of fUML's language definition

Semantics. The semantics of fUML is defined in an operational way by the *fUML execution model*. The fUML execution model is defined using bUML—a subset of fUML—and specifies a *virtual machine* capable of interpreting fUML models. In the fUML standard, the behavior of this virtual machine is represented using Java as a surface notation and a translation of Java into bUML is provided. The semantics of bUML is defined in a translational way using the first-order logic formalism PSL [71]. This is omitted in Figure 3.1. A reference implementation of the fUML virtual machine was implemented by Model Driven Solutions on behalf of the Lockheed Martin Corporation [91], both of whom are submitters of the fUML standard. The reference implementation is developed with Java and available under the Academic Free License version 3.0. The objective of this reference implementation is to encourage tool vendors to implement the fUML standard in their tools and to provide a reference that can assist in evaluating the conformance of vendor implementations with the fUML standard.

In the following sections, we provide a thorough description of the abstract syntax and the behavioral semantics of fUML. Firstly, we present the metamodel of the fUML subset. Secondly, we discuss the fUML execution model and explain how the standardized virtual machine works.

3.2 fUML Subset

The fUML subset comprises the structural kernel of UML (UML package Classes), the behavioral kernel of UML (UML package CommonBehaviors), a major subset of UML's activities sublanguage (UML package Activities), as well as a major subset of UML's action language (UML package Actions). Hence, to define the structure of a system, fUML provides class modeling concepts, and to define the behavior of a system, fUML provides activity modeling concepts. Furthermore, fUML provides its own library called *Foundational Model Library*, which provides primitive data types and primitive behaviors operating on these primitive data types. In the following, we provide a brief overview of the modeling concepts included in fUML.

3.2.1 Structure

fUML includes a subset of the class modeling concepts provided by UML for defining the structure of a system. Figure 3.2 shows an excerpt of fUML's metamodel comprising the core class modeling concepts. Classes (metaclass Class) can own attributes (metaclass Property), which define a type (generalization relation to TypedElement) and a multiplicity (generalization relation to MultiplicityElement). Associations (metaclass Association) define possible links between class instances. The types of the instances, which can be linked via an association, are defined by the association's member ends (reference memberEnd). Furthermore, structured data types (metaclass DataType) can be defined, which can own attributes. Primitive types (metaclass PrimitiveType) are a specialization of data types. fUML provides four predefined primitive types, namely Boolean, Integer, String, and UnlimitedNatural in the Foundational Model Library. Another specialization of data types are enumerations (metaclass Enumeration), which define a set of literals (metaclass EnumerationLiteral).

3.2.2 Behavior

To enable modeling the behavior of a system, fUML includes a subset of the activity modeling concepts provided by UML. As shown in the excerpt of fUML's metamodel depicted in Figure 3.3, classes are behaviored classifiers (generalization relation to BehavioredClassifier) and can, thus, own behaviors (reference ownedBehavior). Behaviors (metaclass Behavior) can be either associated with operations of classes (metaclass Operation) or defined as classifier behaviors of classes. Behaviors associated with operations (reference method inherited by generalization relation to BehavioralFeature) provide an implementation of those operations and must own parameters (metaclass Parameter) matching the parameters owned by the respective operation. The classifier behavior of a class (reference classifierBehavior inherited by generalization relation to BehavioredClassifier) defines a behavior which is called when an instance of the class is created.

Figure 3.4 shows an excerpt of fUML's metamodel comprising the basic concepts for modeling activities, which are the only kind of behavior included in fUML that can be used to model the behavior of a system. Activities (metaclass Activity) consist of activity nodes (metaclass ActivityNode) and activity edges (metaclass ActivityEdge). There are three types of nodes available in fUML, namely actions (metaclass Action), control nodes (metaclass ControlNode), and object nodes (metaclass ObjectNode).

Actions constitute the fundamental unit of executable behavior and their execution represents some processing in the modeled system. fUML contains 27 types of actions, which are predefined by UML's action language. They can be divided into object actions for handling instances of classes (i.e., objects), structural feature actions for handling attribute values of objects, link actions for handling links between objects, and communication actions for invoking activities synchronously or asynchronously. Furthermore, fUML provides structured activity nodes, which are special actions for grouping activity nodes and activity edges (metaclass StructuredActivityNode), expressing alternatives (metaclass ConditionalNode), and expressing loops (metaclasses LoopNode and ExpansionRegion).

Control nodes can be used to define the start and the end of an activity, as well as alternative and concurrent branches of an activity.

Object nodes are used to define the input and output of activities and actions. For defining the input and output of activities, activity parameter nodes (metaclass ActivityParameterNode) are used, which are associated with the respective activity parameters (reference parameter). The input and output of actions are defined with input pins (metaclass InputPin) and output pins (metaclass OutputPin), respectively. Expansion regions are used for iterating over collections of elements provided as input and computing collections of elements as output. Thereby, the input collections and output collections are defined with expansion nodes (metaclass ExpansionNode).

Activity edges are used to connect activity nodes with each other. Control flow edges (metaclass ControlFlow) define the flow of control among activity nodes, whereas object flow edges (metaclass ObjectFlow) denote the flow of data among activity nodes.

A detailed description of the actions and control nodes included in fUML is provided in Appendix A.

3.2.3 Foundational Model Library

fUML provides its own model library consisting of model elements that can be reused in fUML models. The library defines four primitive data types, namely Boolean, Integer, String, and UnlimitedNatural. For these primitive data types, the library provides primitive behaviors also called functions. The provided Boolean functions include Boolean operators, such as AND, OR, XOR, Integer functions include Integer operators, such as addition, subtraction, and multiplication, String functions include String operators, such as string concatenation and substring extraction, and UnlimitedNatural functions include comparison functions and conversion functions to String and Integer. Furthermore, functions on lists are provided by the library, namely for determining the size of a list and retrieving list item at specific indexes. However, no primitive data type has been introduced for lists, because actions can receive multiple input values through input pins with a defined multiplicity greater than one, which constitute a list of values. Besides these primitive data types and primitive behaviors, the Foundational Model Library also provides model elements defining input and output channels, which can be used for receiving input and sending output from an executing fUML model, respectively.

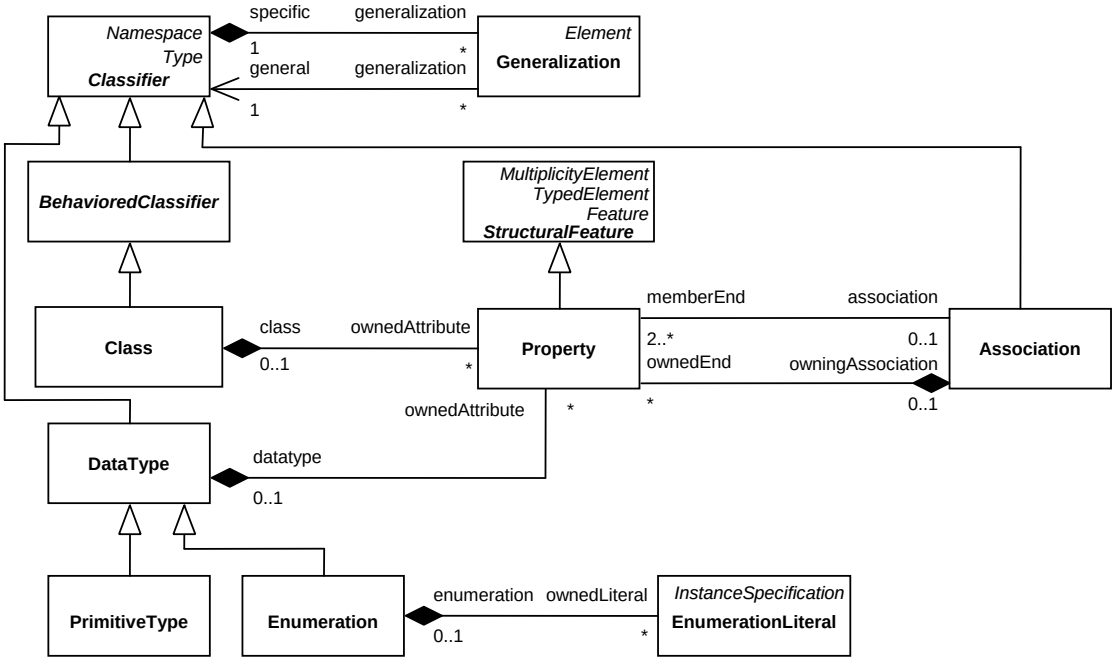


Figure 3.2: Excerpt of the fUML metamodel for modeling the structure of a system

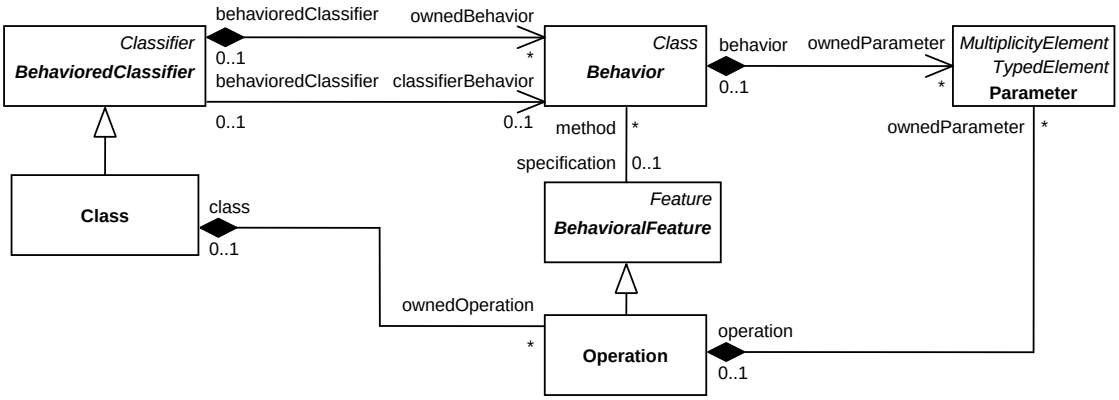


Figure 3.3: Excerpt of the fUML metamodel for modeling the connections between the structure and the behavior of a system

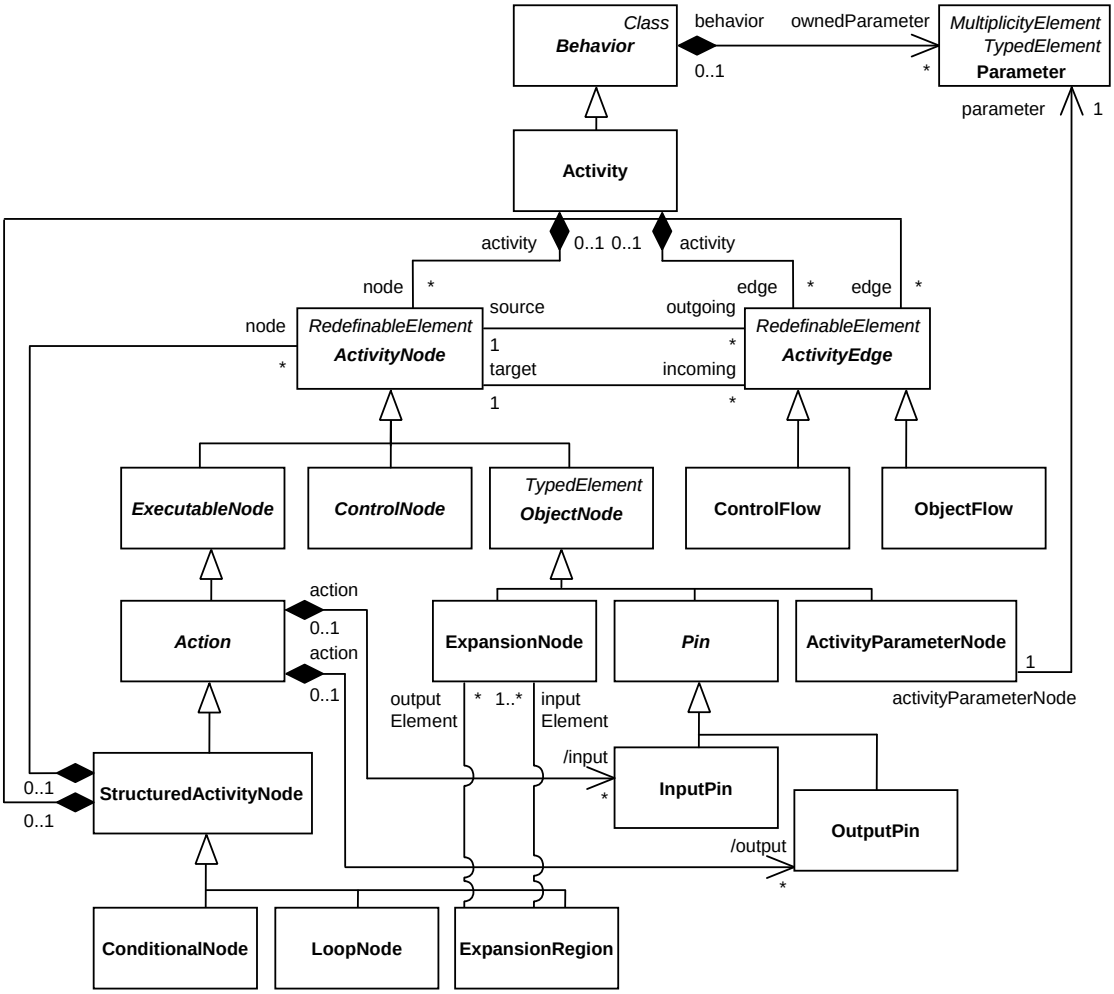


Figure 3.4: Excerpt of the fUML metamodel for modeling the behavior of a system

3.3 fUML Virtual Machine

The behavioral semantics of the fUML subset is defined by the *fUML execution model*, which specifies a *virtual machine* for executing fUML models. Thereby, the fUML execution model is defined with bUML, which is a subset of fUML whose semantics is formally defined in a translational way. In this thesis, we focus on the fUML virtual machine defined by the fUML execution model and described in the following.

To define the behavioral semantics of each modeling concept included in fUML, the visitor pattern is used. This means that for each metaclass defined in the metamodel of fUML a so-called *semantic visitor class* exists, which defines the execution behavior of the respective modeling concept. Thereby, it is distinguished between three types of semantic visitor classes. *Evaluation visitor classes* define how value specifications are evaluated, i.e., they define how values are created from a value specification. For instance, the evaluation visitor class LiteralBooleanEvaluation defines how the specification of a boolean value (metaclass LiteralBoolean) is evaluated to a boolean value (semantic visitor class BooleanValue). *Activation visitor classes* define the semantics of activity nodes, i.e., they define how instances of the metaclass ActivityNode are executed. For instance, the activation visitor class CreateObjectActionActivation defines the semantics of the create object action (metaclass CreateObjectAction). *Execution visitor classes* define the semantics of behaviors, i.e., they define how instances of the metaclass Behavior are executed. For instance, the execution visitor class ActivityExecution defines the semantics of activities (metaclass Activity).

Besides the semantic visitor classes, the fUML virtual machine also defines an *execution environment*. This execution environment is responsible for handling the execution of fUML models. This includes the instantiation of the semantic visitor classes for executing fUML models, the provision of primitive data types and primitive behaviors as defined in the Foundational Model Library, the handling of semantic variation points, and the management of objects created during the model execution.

In the following, we provide an overview of the semantic visitor classes and the execution environment of the fUML virtual machine, and explain the process of executing an fUML model carried out by the fUML virtual machine based on an example. We will refer to the classes defined by the fUML virtual machine as *semantic visitor classes* and to the classes defined by the fUML metamodel as *metaclasses*.

3.3.1 Evaluation Visitors

To specify values in an fUML model, value specifications (metaclass ValueSpecification) are used. According to the primitive data types provided by fUML, value specifications exist for specifying Boolean values (metaclass LiteralBoolean), Integer values (metaclass LiteralInteger), String values (metaclass LiteralString), and UnlimitedNatural values (metaclass LiteralUnlimitedNatural), as well as Null values (metaclass LiteralNull). Furthermore, instances of enumerations, data types, and classes can be specified (metaclasses InstanceValue and InstanceSpecification).

The evaluation visitor classes define how values are created from such value specifications. For each type of value specification, a dedicated evaluation visitor class exists, as depicted in Figure 3.5. For evaluating Boolean, Integer, String, UnlimitedNatural, and Null val-

ues, the evaluation visitor classes LiteralBooleanEvaluation, LiteralIntegerEvaluation, LiteralStringEvaluation, LiteralUnlimitedNaturalEvaluation, and LiteralNullEvaluation are used, respectively. The evaluation visitor class InstanceValueEvaluation evaluates the specification of enumeration, data type, and class instances.

The evaluation of a value specification is done by the operation evaluate() and results in a value. Values are defined by the semantic visitor class Value, whose subtypes are depicted in Figure 3.6. Boolean, Integer, String, and UnlimitedNatural values are represented by instances of the semantic visitor classes BooleanValue, IntegerValue, StringValue, and UnlimitedNaturalValue, respectively. Enumeration values are instances of the semantic visitor class EnumerationValue and refer to the enumeration literal they carry. Instances of data types, classes, and associations are represented by instances of the semantic visitor classes DataValue, Object, and Link, respectively. They own feature values (semantic visitor class FeatureValue), which define the values assigned to the structural features of their respective type, i.e., the structural features defined by the respective data type, class, or association. The values assigned to those structural features are owned by the respective feature value. Objects are always carried by references (semantic visitor class Reference).

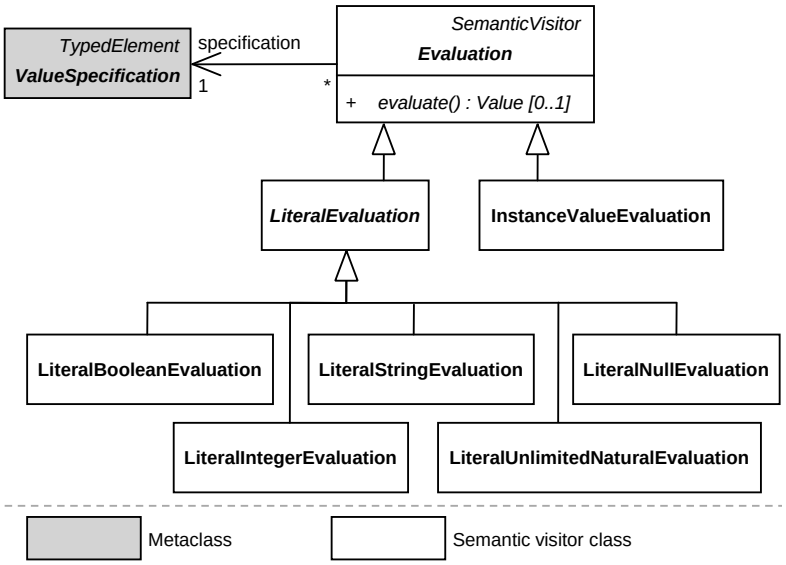


Figure 3.5: Evaluation visitor classes of the fUML virtual machine

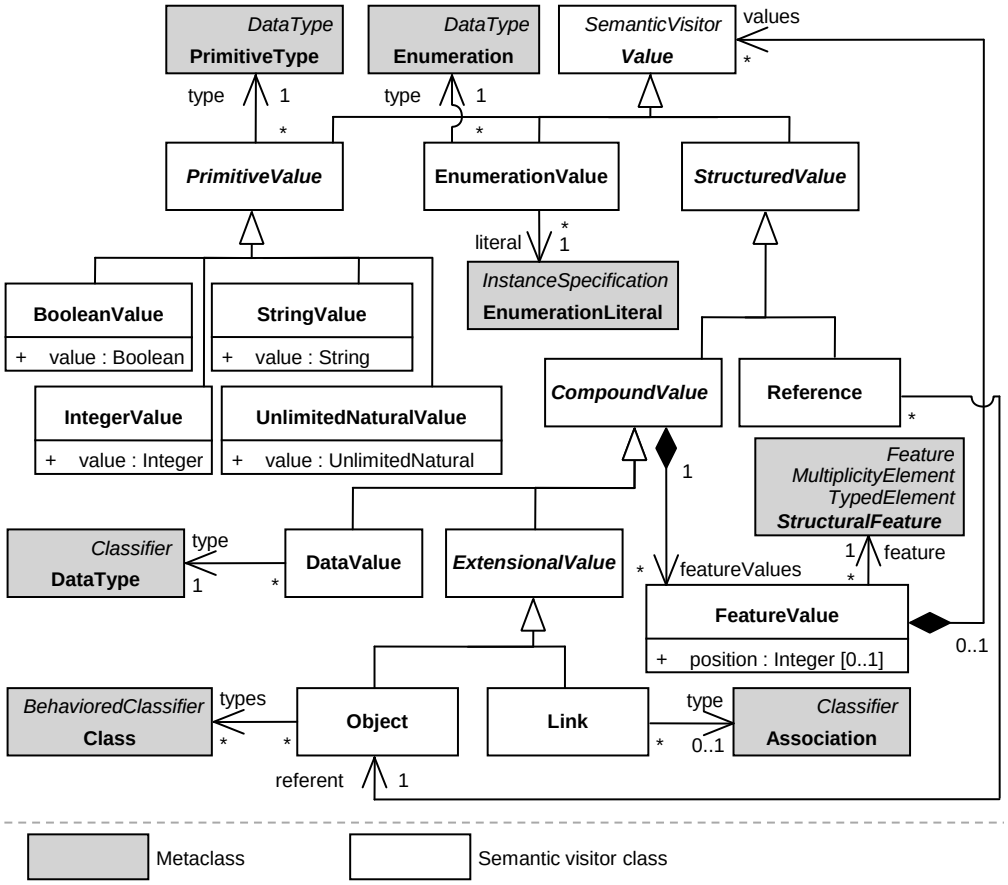


Figure 3.6: Semantic visitor classes of the fUML virtual machine for representing values

3.3.2 Activation Visitors

The activation visitor classes of the fUML virtual machine define the behavior of activity nodes. The activation visitor class ActivityNodeActivation shown in Figure 3.7 constitutes the superclass of all activation visitor classes defined by the fUML virtual machine. For each type of activity node, a subclass of ActivityNodeActivation exists, which defines how an activity node of this type is executed. Besides defining *how* an activity node is executed, the activation visitor classes also define *when* an activity node is executed. This definition is based on the definition of token flow semantics similar to the token flow semantics of Petri nets. Informally speaking, an activity node is executed, when all required control tokens are available through incoming control flow edges and all required object tokens are available through incoming object flow edges. After the execution of an activity node, control tokens and object tokens are offered to the successor nodes via outgoing control flow edges and outgoing object flow edges, respectively.

The semantic visitor class Token and its subclasses ControlToken and ObjectToken depicted in Figure 3.7 define the behavior of tokens as well as how tokens are represented during execution. Tokens are always owned by the activity node activation offering the tokens via activity edges to successor nodes. How tokens are offered through activity edges is defined by the semantic visitor class ActivityEdgeInstance. ActivityNodeActivation instances are connected via ActivityEdgeInstances according to the connection of the respective activity nodes via activity edges defined in the executed fUML model. The structure and behavior of token offers themselves are defined by the semantic visitor class Offer.

The activation visitor class ObjectNodeActivation defines how tokens are handled by object nodes. Activity parameter nodes, whose behavior is defined by ActivityParameterNodeActivation, are responsible for providing input values to the activity execution and receiving output values from the activity execution. The behavior of input pins and output pins is defined by the activation visitor classes InputPinActivation and OutputPinActivation. While input pins receive values from incoming object flow edges and pass them to the associated actions, output pins receive values from the associated actions and pass them to outgoing object flow edges.

How actions are executed is defined by the activation visitor class ActionActivation and its subclasses. They are responsible for executing some kind of behavior, which is defined by the operation doAction(), and for providing object tokens to their output pins.

Control nodes basically route incoming tokens according to some rules to outgoing activity edges. These routing rules are defined by the subclasses of the activation visitor class ControlNodeActivation through the implementation of the operation fire() .

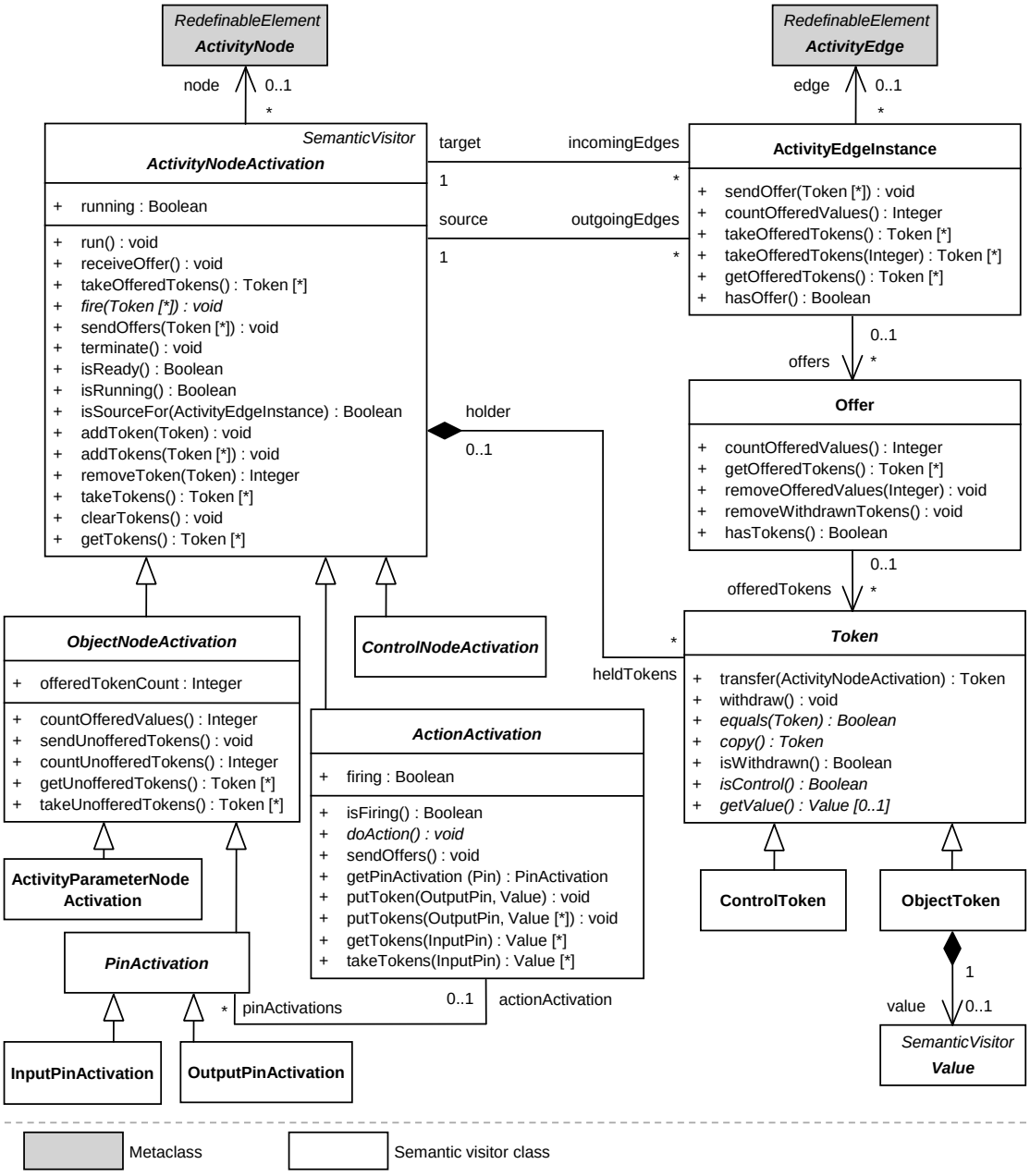


Figure 3.7: Activation visitor classes of the fUML virtual machine

3.3.3 Execution Visitors

The execution visitor classes of the fUML virtual machine depicted in Figure 3.8 define how behaviors—in particular activities—are executed. This definition is given by the implementations of the operation execute() specified by the most basic execution visitor class Execution. For executing a behavior, the execution is provided with input parameter values. After executing a behavior, the execution may provide output parameter values. Both, input parameter values and output parameter values are defined by the semantic visitor class ParameterValue.

The execution visitor class ActivityExecution defines how activities are executed. When the execution of an activity starts, it first creates an instance of the semantic visitor class ActivityNodeActivationGroup, which in turn instantiates the activation visitor classes for the nodes owned by the activity and the ActivityEdgeInstance class for the edges owned by the activity. In case the activity contains structured activity nodes, the activation visitor class StructuredActivityNodeActivation is instantiated for each of these structured activity nodes. The StructuredActivityNodeActivation instances subsequently also create an instance of the semantic visitor class ActivityNodeActivationGroup, which is responsible for instantiating the activation visitor classes for the nodes and edges owned be the respective structured activity node. The purpose of the semantic visitor class ActivityNodeActivationGroup is to group activity node activations for nodes owned by the activity and for nodes owned by structured activity nodes of activities. After the instantiation of the activation visitor classes, tokens are provided to the initially enabled nodes of the activity, which are initial nodes, input activity parameter nodes, and actions without incoming control flow edges and without input pins. Subsequently, these nodes start executing, leading to successor nodes being enabled and executed. The activity execution terminates, when no activity node is enabled anymore or when an activity final node has been executed. After the termination, the activity execution collects the object tokens residing on output activity parameter nodes and provides them as output.

The execution visitor class OpaqueBehaviorExecution defines how opaque behaviors are executed, which are used to define the primitive behaviors provided by the Foundational Model Library. For defining how a primitive behavior is executed, the operation doBody() is implemented by concrete subclasses of OpaqueBehaviorExecution. It takes as input the values provided for the input parameters of the primitive behavior, as well as an initialized set of parameter values for the output parameters of the primitive behavior. To provide the output of the primitive behavior, the initialized output parameter values are modified by the respective implementation. For instance, the execution visitor class BooleanAndFunctionBehaviorExecution takes as input two Boolean values, and returns the result of applying the Boolean AND operator.

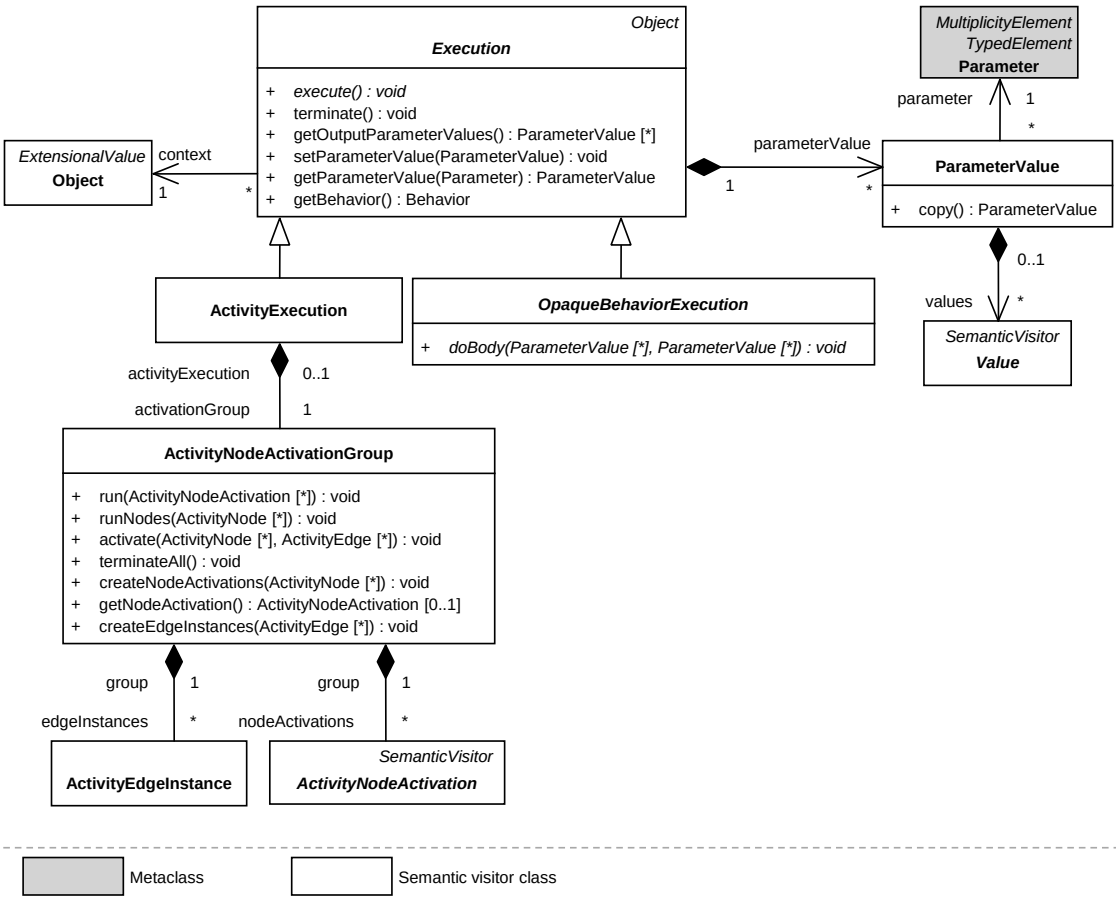


Figure 3.8: Execution visitor classes of the fUML virtual machine

3.3.4 Execution Environment

The execution environment of the fUML virtual machine depicted in Figure 3.9 is responsible for managing the execution of an fUML model. The class Executor serves as interface of the fUML virtual machine, which can be used to execute fUML models. Therefore, it provides the following three operations. The operation execute() can be used for synchronously starting the execution of a behavior defined in the fUML model. Thereby input parameter values can be provided to the execution of the behavior and a context object can be defined, which is accessible by the behavior during execution. This operation returns output parameter values resulting from the behavior execution. The operation start() asynchronously starts the execution of a behavior. Again, input parameter values can be provided for the execution. This operation returns a reference to the instance of the executing behavior (i.e., an instance of the execution visitor class Execution), which can be used to obtain output parameter values of the behavior execution after its termination. The operation evaluate() can be used to evaluate a value specification and obtain the resulting value.

Every execution takes place at a locus (environment class Locus), which represents the place where an fUML model is executed. All extensional values, i.e., objects and links, created during the execution of an fUML models are persisted at the locus. The locus may contain extensional values created by prior executions or by the environment itself.

The locus also provides a factory (environment class ExecutionFactory), which enables the executor at the locus to instantiate semantic visitor classes for the execution of an fUML model, which are in the end responsible for the execution. This factory also provides the available primitive data types and primitive behaviors as defined in the Foundational Model Library. For the provided primitive behaviors, the factory maintains corresponding execution instances (i.e., instances of the execution visitor class OpaqueBehaviorExecution) serving as prototypes, which are looked-up when primitive behaviors are called via call behavior actions, copied, and executed. Furthermore, the factory maintains implementations of semantic variation points defined by fUML, which are the dispatching of events caused by send signal actions, the dispatching of polymorphic operation calls caused by call operation actions, and choosing between alternative paths during execution leading to nondeterminism caused for instance by add structural feature value actions when values are added to an object for a multi-valued structural feature without specifying the position at which the value should be inserted. Implementations for these semantic variation points can be provided by implementing the abstract strategy classes GetNextEventStrategy, DispatchStrategy, and ChoiceStrategy, and adding them to the execution factory.

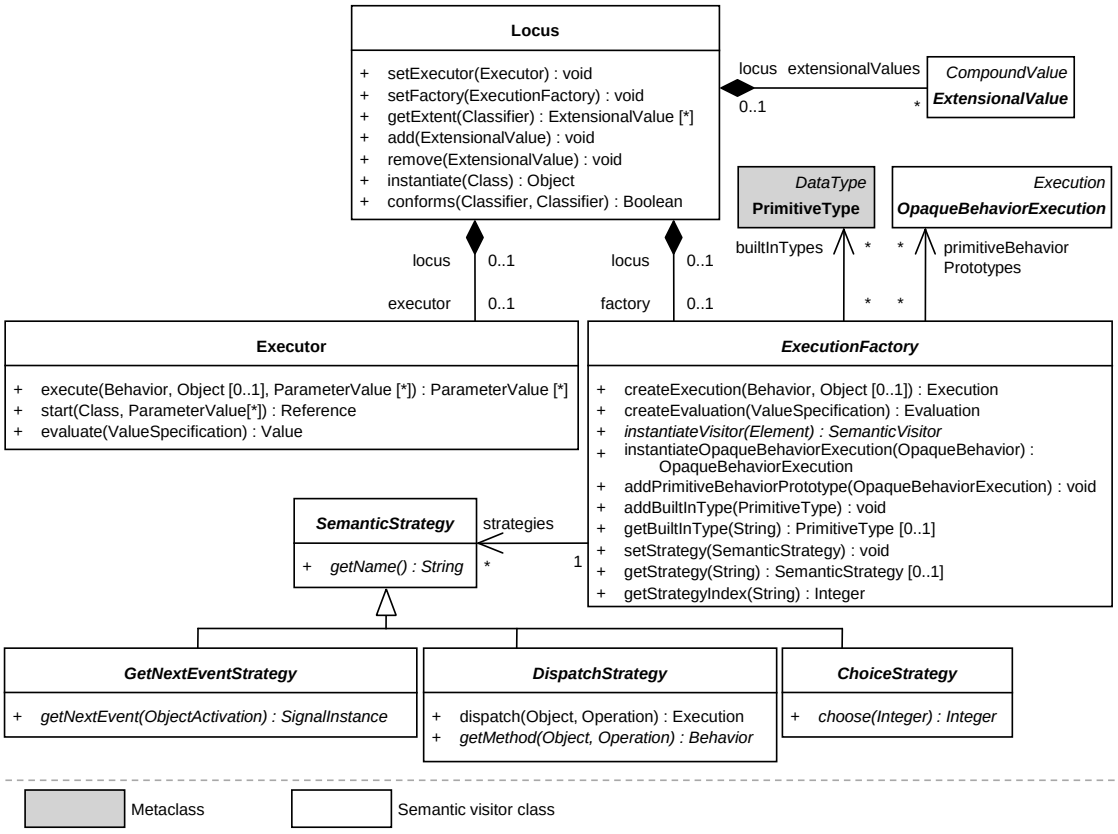


Figure 3.9: Execution environment classes of the fUML virtual machine

3.3.5 Example

To illustrate the functionality of the fUML virtual machine, we consider the execution of the activity depicted in Figure 3.10. This activity consists of an initial node, which constitutes the starting point of the activity, and a subsequent create object action, which instantiates a classifier and provides the resulting object to the output activity parameter node.

How the execution of this activity is carried out by the fUML virtual machine is depicted in Figure 3.11, which shows the operation calls among the execution model elements instantiated for executing the activity. Figure 3.12 shows the execution model in greater detail as well as its evolution during the execution.

As shown in Figure 3.11, to start the execution of the activity, the operation execute() of the executor `e` is called providing the activity as input to the operation call. As can be seen in Figure 3.12, the executor `e` resides at the locus `l`, which provides the execution factory `ef`. This execution factory is used by the executor to create an activity execution `activity_exe` by calling the operation createExecution().

After the creation of the activity execution, the executor calls its operation execute(). As a result of this operation call, the activity execution creates an activity node activation group `group` and calls its operation activate(). Consequently, the activity node activation group instantiates the activation visitor classes for the activity nodes contained by the executing activity by calling the operation createNodeActivations() and the activity edge instance class for the activity edges by calling the operation createEdgeInstances(). As can also be seen in Figure 3.12, after the operation createEdgeInstances() has been executed (time t1), the locus `l` contains the activity execution `activity_exe` as extensional value, which contains the activity node activation group `group` consisting of four activity node activations, namely the initial node activation `init_exe`, the create object action activation `action_exe`, the output pin activation `result_exe`, and the activity parameter node activation `out_exe`, as well as two activity edge instances `e1_exe` and `e2_exe`. They are linked to each other according to the references defined between the activity nodes and activity edges of the executing activity and refer to the respective model element, i.e., to the respective activity node or activity edge.

After the creation of the activity node activations and activity edge instances, the activity node activation group executes the operation run(), which starts the execution of the initial node by invoking the operation receiveOffer() of the initial node activation `init_exe`. Please note that no actual token is provided to `init_exe`. The call of the operation receiveOffer() causes the initial node activation to call the operation fire(), which creates a control token. This control token is sent to the subsequent node by calling the operation sendOffers(), which in turn calls the operation sendOffer() of the outgoing activity edge instance `e1_exe`. The operation sendOffer() creates a token offer for the target activity node activation, which is the create object action activation `action_exe`. At this point of the execution (time t2), the initial node activation `init_exe` holds one control token `t1`, and the activity edge instance `e1_exe` provides one token offer `o1` offering this control token as depicted in Figure 3.12.

To start the execution of the create object action, the operation receiveOffer() of the create object activation `action_exe` is called. This causes the call of the operation fire(), which in turn calls the operation doAction(). In the doAction() operation, the classifier specified by the create object action is instantiated resulting in the object `obj1` having this classifier set as its

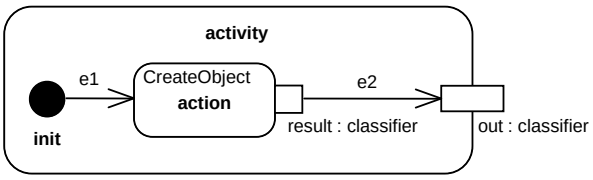


Figure 3.10: fUML virtual machine example: Activity

type. Furthermore, the reference `r1` referring to this object is created. Both the object and the reference to this object are depicted in Figure 3.12 (time t3).

After the instantiation of the classifier, an object token is created for passing the created reference `r1` to the succeeding activity parameter node. To send this object token to the activity parameter node, the operation sendOffers() of the create object action activation `action_exe` is called. This operation calls the operation sendUnofferedTokens() of the output pin activation `result_exe`, which in turns calls the operation sendOffers() resulting in the call of the operation sendOffer() of the outgoing activity edge instance `e2_exe`. The last operation call causes the creation of a token offer for the previously created object token. As can be seen in Figure 3.12, at this point of the activity execution (time t4), the output pin activation `result_exe` holds the object token `t2`, which carries the reference `r1`, which refers to the object `obj1`. Furthermore, the activity edge instance `e2_exe` provides the offer `o2` offering this object token.

The offer of the object token `t2` is received by the operation receiveOffer() of the activity parameter node activation `out_exe`. Subsequently, the operation fire() of the activity parameter node activation is called. This operation adds a new object token to the activity parameter node activation carrying the reference `r1` transported by the received object token `t2`. This constitutes the termination of the activity execution, because no more tokens can be sent and, consequently, no activity node can be executed anymore. As last step done by the activity execution `activity_exe`, it calls the operation getTokens() of the activity parameter node activation for retrieving the newly added object token and creates an output parameter value holding the reference `r1`. As depicted in Figure 3.12, at this point of the execution (time t5), the activity parameter node activation `out_exe` holds the new created object token `t3` carrying the reference `r1` and the activity execution `activity_exe` contains the created parameter value `pv1` for this reference.

Finally, the executor `e` retrieves the parameter value `pv1` from the activity execution `activity_exe` by calling the operation getOutputParameterValues(). This parameter value constitutes the final output of the activity execution and is provided as return value of the operation execute() of the executor. Before the output is provided, the activity execution is destroyed by the executor, resulting in the destruction of activity node activation group, as well as the activity node activations, activity edge instances, and the remaining token `t3`. Please note that the token `t1` and the offer `o1` as well as the token `t2` and the offer `o2` have been destroyed in the course of executing the create object action and the activity parameter node, respectively. After the execution finished, only the created object `obj1` remains at the locus and is available to subsequent executions.

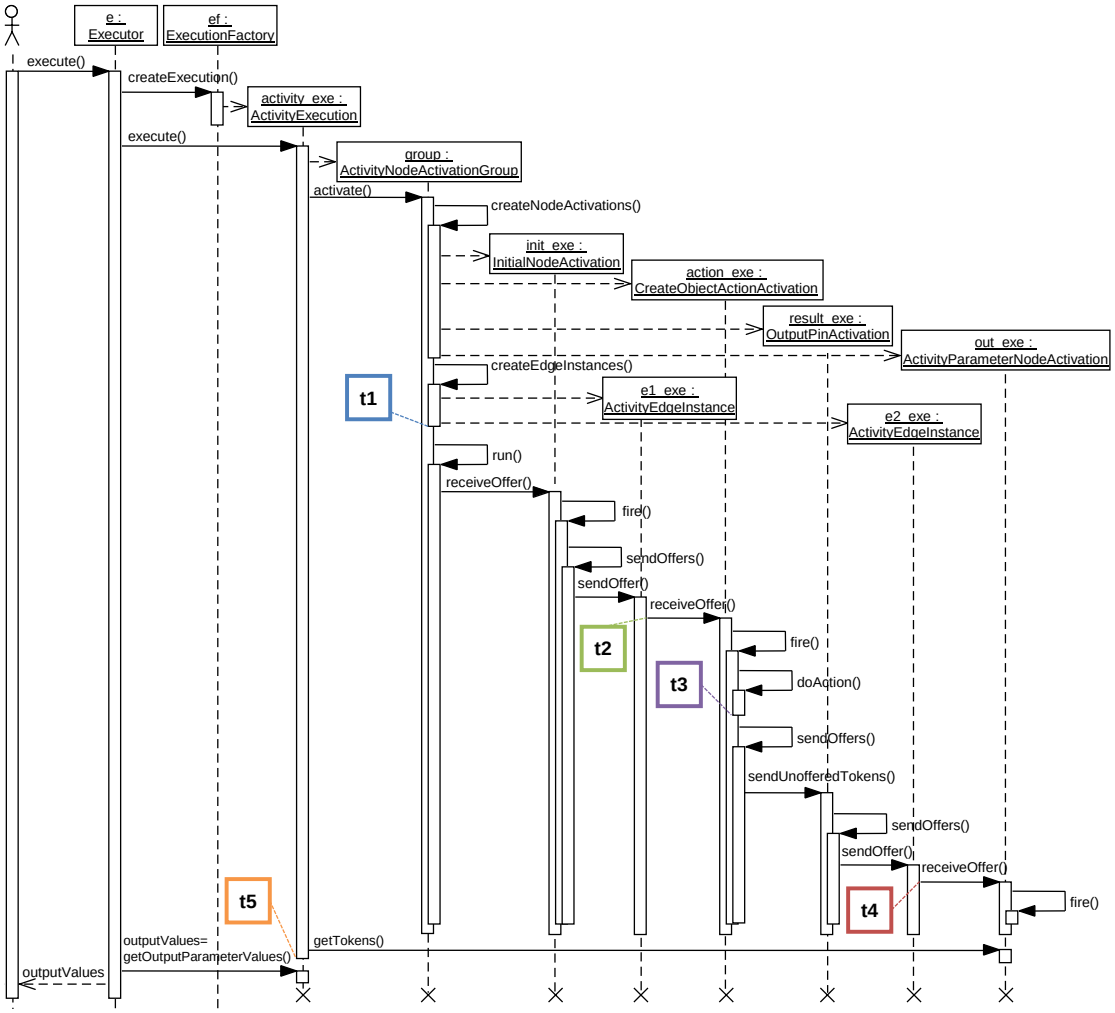


Figure 3.11: fUML virtual machine example: Calls among execution model elements

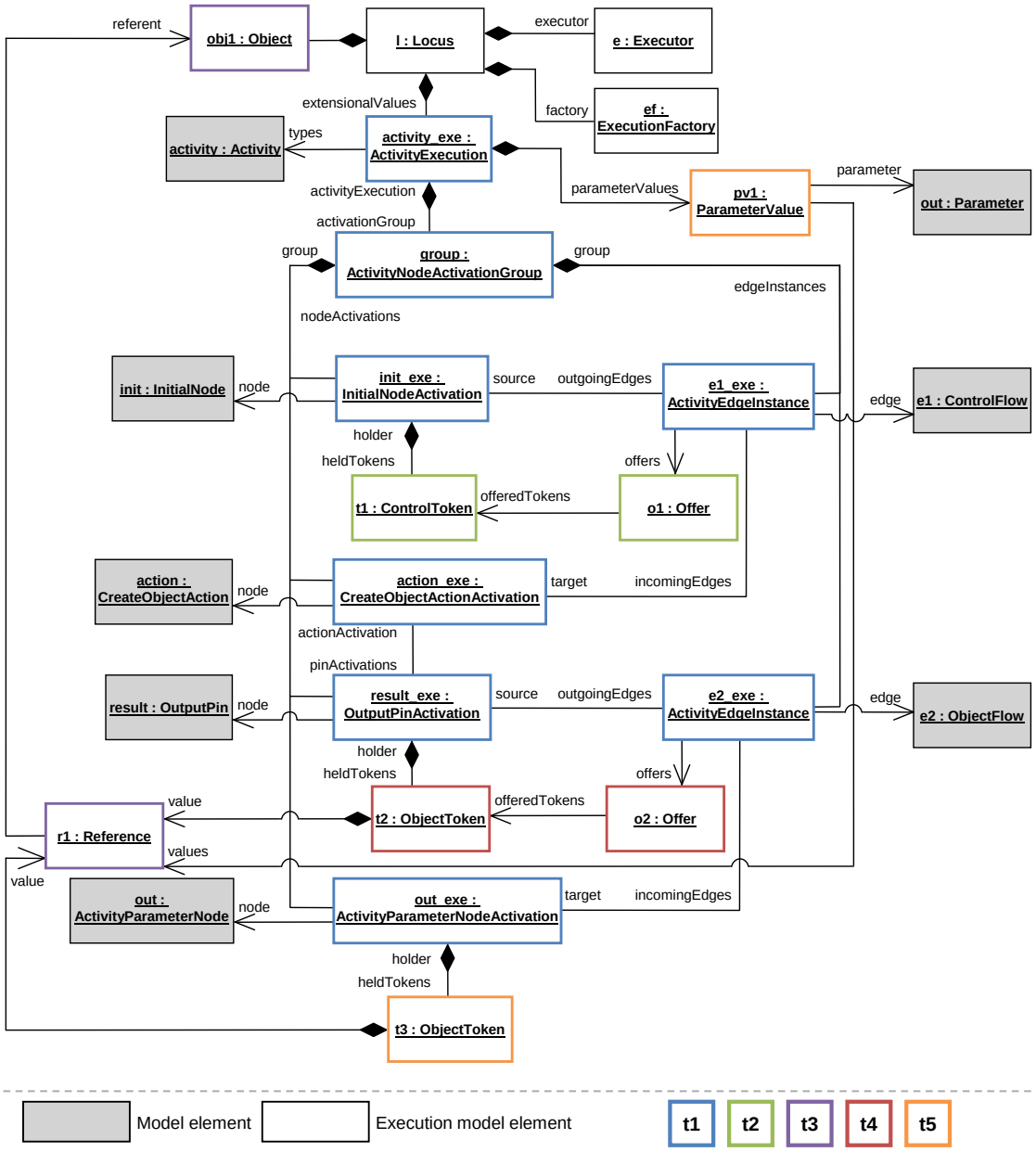


Figure 3.12: fUML virtual machine example: Execution model

CHAPTER 4

Extensions of the fUML Execution Environment

4.1 Design Rationale

The fUML standard precisely defines the behavioral semantics of a selected subset of UML. The behavioral semantics is defined in terms of a virtual machine capable of executing UML models, which conform to the selected subset of UML. As described in Chapter 3 and depicted on the left-hand side of Figure 4.1, 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.

With the introduction of fUML, UML evolves to an executable modeling language possessing a standardized behavioral semantics specifications. This constitutes a major step towards the utilization of executable UML models. As discussed in Section 2.1, one of the main advantages of executable models is that they can be analyzed for validation, verification, and comprehension purposes starting from the early phases of the software development process. Therewith, the quality of software systems developed based on executable models can be increased. Methods proposed for this purpose comprise testing, formal analysis, dynamic analysis, debugging, and non-functional property analysis. The need for implementations of these analysis methods for fUML becomes even more evident, if we bear in mind that fUML can be used as a programming language for completely building executable systems [140], meaning that fUML models could replace source code and be directly deployed in a production environment. In such an application scenario, it is apparent that rich IDE support for fUML comprising for instance debuggers and testing environments—as known from current IDEs for source code development—are required. However, the full potential of fUML models cannot be exploited yet, because the standardized fUML execution environment comprising the fUML virtual machine does not provide the means for implementing these analysis methods. In particular, it does not exhibit the required characteristics observability, controllability, and analyzability, discussed in the following.

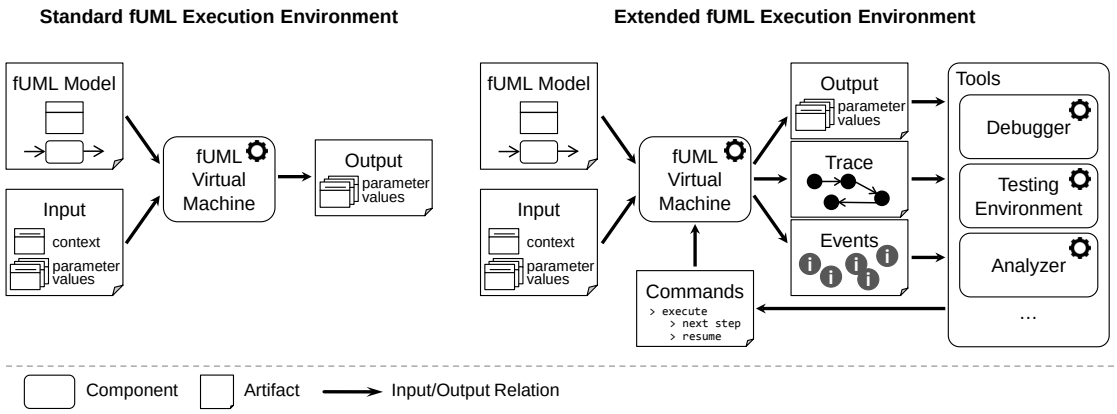


Figure 4.1: Overview of fUML execution environment extensions

Observability. An important characteristic of a virtual machine is observability, that is the ability to monitor the state of an execution being carried out by the virtual machine. Observability is a prerequisite for implementing important analysis methods based on execution capabilities, including, for instance, debugging, non-functional property analysis through simulation, profiling, and monitoring. However, the fUML virtual machine does not enable the observation of the execution of a model during runtime. Thus, it is not possible to retrieve the current position of the execution and the current state of existing objects.

Controllability. Complementary to observability is controllability, that is the ability to control executions being carried out by a virtual machine. Controllability constitutes a crucial basis for implementing methods and techniques again for debugging and non-functional property analysis through simulation, as well as similar methods utilizing the executability of models. The fUML virtual machine, however, does not provide any support for controlling the execution of an fUML model, such as support for suspending and resuming ongoing executions at a particular position.

Analyzability. The characteristic of analyzability—more specifically dynamic analyzability—is the ability to analyze ongoing or completed executions based on captured runtime information. As discussed in Section 2.1.3, dynamic analysis is particularly useful for programs developed with object-oriented languages—as fUML is. Applications of dynamic analysis include comprehension, analysis of non-functional properties (e.g., performance analysis), testing (e.g., analysis of test coverage), and evolution (e.g., version differencing). However, the fUML virtual machine does not provide runtime information about a model execution other than output parameter values. Missing runtime information includes, for instance, information about which part of a model has been executed and which objects have been created or manipulated during the execution.

To overcome these limitations, and, hence, build the basis for implementing methods and tech-

niques for analyzing executable UML models, we extended the fUML execution environment [97]. An overview of the extensions is depicted on the right-hand side of Figure 4.1. The developed extensions comprise an *event mechanism*, which issues events notifying about state changes of an ongoing model execution and, hence, enables the observation of ongoing executions. Furthermore, we integrated a *command interface* into the fUML execution environment, which enables the issuance of commands for controlling the execution of a model, in particular, to suspend and resume the execution, as well as to step through the execution model element per model element. To provide the ability to dynamically analyze a partially or completely performed execution of a model, we elaborated a *trace model* tailored to UML activities for capturing execution traces, and we extended the fUML execution environment to record a trace during the execution of a model.

Standard conformance. One important overall objective during the development of the extensions of the fUML execution environment was to ensure the conformance with the fUML standard. In particular, this means that the behavior of the fUML virtual machine should not be modified by incorporating the extensions and the behavioral semantics of UML defined in the fUML standard should be retained. Hence, we decided against building our own virtual machine and instead incorporated our extensions into the reference implementation of the fUML virtual machine [91]. Furthermore, we decided not to modify the source code of the reference implementation directly, but instead to use the aspect-oriented programming language AspectJ [27] to weave our extensions into the reference implementation. A resulting advantage of this approach is that we can directly adopt bug fixes applied to the reference implementation, as well as modifications due to releases of newer versions of the fUML standard[1].

4.2 Event Mechanism

To equip the fUML virtual machine with the characteristic of *observability*, we introduced an *event mechanism* into the fUML virtual machine. The event mechanism is responsible for detecting *state changes* of an execution currently being executed by the fUML virtual machine and notifying about these state changes by issuing corresponding *events*. In particular, the following types of state changes are detected by the event mechanism and reported through the issuing of events.

- **Position of execution.** Dedicated events inform about the current position of an ongoing execution. Thereby, the current position of an ongoing execution constitutes the last executed model element, i.e., the model element, which has been executed in the last execution step.
- **State of values.** Dedicated events inform about modifications of extensional values residing at the locus of an ongoing execution. As described in Section 3.3.4, every execution carried out by the fUML virtual machine takes place at a locus. Extensional values, i.e.,

[1] This thesis builds upon version 1.0 of the fUML standard [114]. Version 1.1 [119] is the most current version of the fUML standard at the time of writing (cf. Section 2.3.2).

objects and links, created by an execution are persisted at the respective locus and are accessible by executions being carried out at the same locus. Hence, modifications of extensional values residing at a specific locus are relevant to each execution being carried out at this locus.

These state changes are reported by the event mechanism with different types of events. The current position of an ongoing execution is reported with *trace events*. Modifications of extensional values residing at the locus of the execution are reported with *extensional value events*.

To detect state changes, we implemented dedicated aspects for these two types of state changes using AspectJ. These aspects define pointcuts for the fUML virtual machine, which determine join points during the execution of an fUML model that constitute state changes of the execution. The advices associated with these pointcuts are responsible for creating respective events and issuing them.

Trace events. Trace events enable the observation of the progress of the execution of an fUML model by reporting on the current position of the execution. The position of an execution is determined by the activities being currently executed, as well as by the executed activity nodes contained by these activities. The different types of trace events are depicted in Figure 4.2. We distinguish between activity events, activity node events, and suspension events.

Activity events (event class ActivityEvent) report on the progress of executing activities. Activity entry events (event class ActivityEntryEvent) indicate the start of executing an activity, while activity exit events (event class ActivityExitEvent) indicate the completion of executing an activity.

Activity node events (event class ActivityNodeEvents) report on the progress of executing activity nodes contained by executing activities. The start of executing an activity node is indicated by an activity node entry event (event class ActivityNodeEntryEvent). The completion of executing an activity node is indicated by an activity node exit event (event class ActivityNodeExitEvent). Thereby, activity node events are triggered for reporting on the progress of executing actions and control nodes. For the execution of object nodes, no activity node events are issued, because they are always executed in the course of executing the associated action or activity. In particular, when starting the execution of an action, first all input pins of the action are executed. Likewise, before completing the execution of an action, all output pins of the action are executed. The same is true for expansion nodes of expansion regions. Expansion nodes taking input for the expansion region are executed first when starting the execution of the expansion region. Expansion nodes providing output of the expansion region are executed before the execution of the expansion region finishes. Furthermore, the execution of input activity parameter nodes and output activity parameter nodes of an activity is always associated with starting and finishing the execution of the activity, respectively.

Suspension events (event class SuspendEvent) report on the suspension of an execution. An execution is suspended in the following two cases. Firstly, if the model is executed stepwise and an execution step was completed, the execution is suspended and a suspend event (event class SuspendEvent) is issued. An execution step comprises either the start of executing an activity, or the completion of executing an action or a control node. Secondly, if the model is executed and a breakpoint is hit, the execution is also suspended and a breakpoint event (event

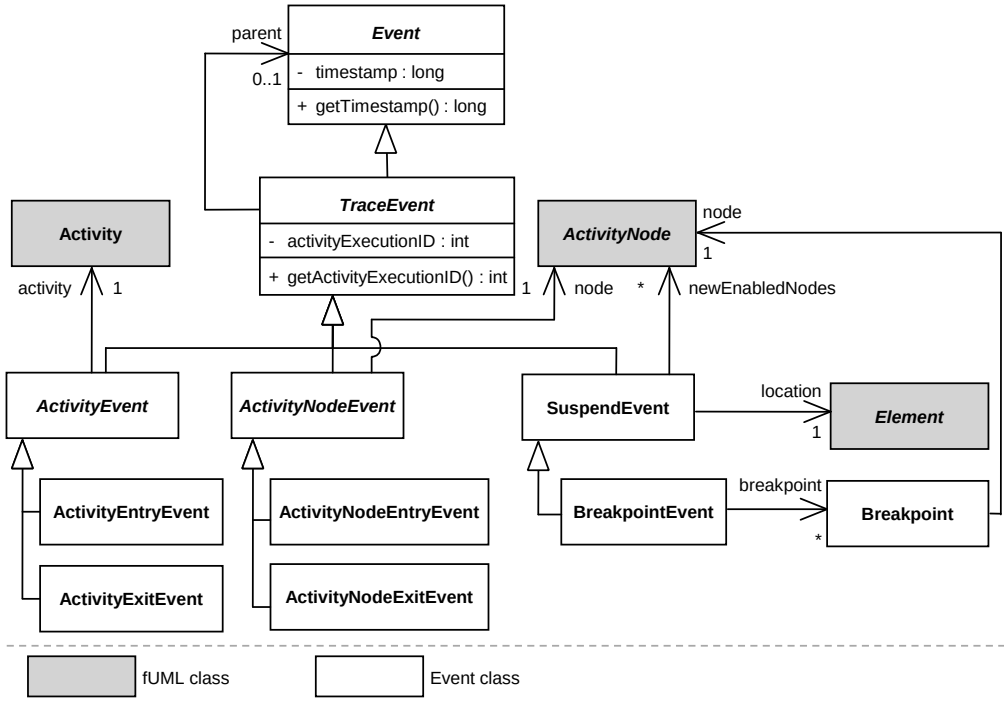


Figure 4.2: Trace events

class BreakpointEvent) referring to the hit breakpoint (reference breakpoint) is issued. Thereby, breakpoints denote that the execution should suspend if a specific action or control node can be executed in the next execution step. Suspension events indicate the location of the suspension (reference location), which is determined by the last executed model element. This is either the executing activity if the execution was suspended after the execution of the activity started but before any action or control node was executed, or an action or control node if the execution was suspended after the execution of the respective action or control node. Furthermore, suspension events indicate which actions and control nodes were enabled due to the last execution step, and, hence, can be executed in the next execution step (reference newEnabledNodes).

To indicate which trace events are associated with a specific activity execution, they provide a unique identifier of this specific activity execution (attribute activityExecutionID). This is important, because the same activity might be executed multiple times during the execution of an fUML model. Furthermore, the relationship between trace events is explicitly captured (reference parent). In particular, exit events refer to the corresponding entry event. For instance, the activity exit event denoting the completion of an activity execution refers to the activity entry event denoting the start of this activity execution. Moreover, activity node entry events always refer to the activity entry event indicating the start of the activity execution that triggered the execution of the respective activity node. Similarly, suspension events refer to the activity entry event indicating the start of the activity execution which triggered the execution of the last executed model element, i.e., the location of the suspension. For activity entry events, we can

distinguish between two cases. If the respective activity execution resulted from the execution of a call action, the activity entry event refers to the activity node entry event indicating the start of executing this call action. Otherwise, if the activity execution constitutes the execution of the starting point activity of the fUML model provided as input to the execute() operation of the fUML executor, no parent is set for the activity entry event.

Extensional value events. Extensional value events enable the observation of modifications of extensional values residing at the locus of the execution of an fUML model. These extensional values comprise objects and links created during model executions carried out at this locus and they are accessible by all model executions being carried out at this locus. For instance, read extent actions can be used to retrieve all instances of a specific classifier, which are residing at the locus of the execution. The different types of extensional value events are depicted in Figure 4.3. We distinguish between extensional value events and feature value events.

Extensional value events (event class ExtensionalValueEvent) report on the construction and destruction of extensional values, as well as on the modification of the type of an extensional value. An extensional value event provides the information about which extensional value was affected (reference value), and which kind of modification was conducted (attribute type). In particular, the creation of an extensional value, for instance caused by the execution of a create object action or create link action, is reported by an extensional value event of the type CREATION. The destruction of an extensional value, for instance caused by the execution of a destroy object action or destroy link action, is reported by an extensional value event of the type DESTRUCTION. The modification of an extensional value's type, which can be caused by the execution of a reclassify object action, is reported by an extensional value event of the type TYPE_ADDED in case a type was added to the extensional value or of the type TYPE_REMOVED in case a type was removed from the extensional value.

Feature value events (event class FeatureValueEvent) report on the construction, destruction, and modification of feature values of extensional values. In the case of objects, feature values constitute the attribute values of the objects. For each attribute owned or inherited by the class constituting the type of an object, the object owns one feature value, which refers to the respective attribute and contains the values assigned to this attribute. In the case of links, feature values constitute references to the linked objects. For each member end of the association constituting the type of the link, the link owns one feature value, which refers to the respective member end and contains a reference to the linked object. Feature value events refer to the affected feature value (reference featureValue) as well as the structural feature of the feature value (reference feature), i.e., the attribute or member end. If a feature value is created for an extensional value or if a feature value of an extensional value is destroyed, a feature value event of the type VALUE_CREATION or VALUE_DESTRUCTION is issued, respectively. This might happen in the course of executing a reclassify object action. If a value is added to or removed from the feature value of an extensional value, for instance caused by the execution of an add structural feature value action or remove structural feature value action, a feature value event of the type VALUE_ADDED or VALUE_REMOVED is issued, respectively. The respective feature value event refers also to the affected values of the feature value (reference values), as well as the position of the affected values (attribute position).

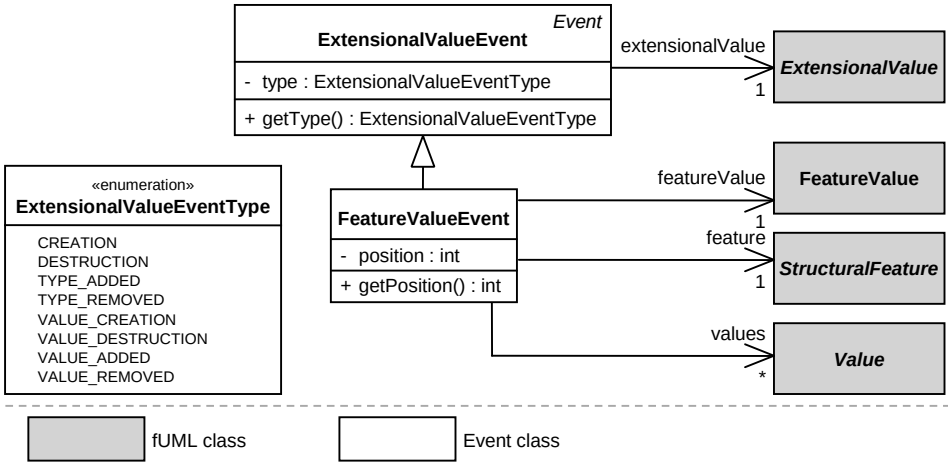


Figure 4.3: Extensional value events

Implementation. To detect state changes of a model execution being carried out by the fUML virtual machine and issue corresponding events, we implemented aspects for the fUML virtual machine using AspectJ. These aspects define pointcuts, which select events in the control flow of the fUML virtual machine that cause relevant state changes of the model execution. The advices associated with these pointcuts are responsible for creating and issuing respective events. A subset of the defined pointcuts and advices is shown in Listing 4.1.

The pointcut activityExecution() shown in Listing 4.1 in line 1 is used to detect the start and the end of an activity execution. The execution of an activity starts when the operation execute() of an activity execution (i.e., an instance of the execution visitor class ActivityExecution) starts executing. Likewise, the execution of an activity is finished when the execution of the same operation finishes. In the first case, an activity entry event is created and issued (cf. before advice shown in line 5 of Listing 4.1). In the second case, an activity exit event is created and issued (cf. after advice shown in line 9 of Listing 4.1).

Keeping track of the execution of actions works similar to keeping track of the execution of activities. Therefore, the pointcut actionExecution() shown in Listing 4.1 in line 13 is used. It detects the start and the end of the execution of actions, which are denoted by the start and the end of executing the operation fire() of an action activation (i.e., an instance of the activation visitor class ActionActivation), respectively. Thus, before and after the execution of the operation fire(), an activity node entry event and an activity node exit event are created and issued, respectively (cf. before and after advices shown in line 17 and 21 of Listing 4.1).

The last pointcut extensionalValueAddedToLocus() shown in Listing 4.1 in line 25 detects the creation of an extensional value at the locus of an ongoing execution. It constitutes a call pointcut for the operation add() of the fUML environment class Locus, which is responsible for adding a newly created extensional value to a specific locus. After each call of this operation, an extensional value event of the type CREATION is created and issued (cf. after advice shown in line 29 of Listing 4.1).

We defined similar additional pointcuts and advices for the fUML virtual machine to com-

```
1  pointcut activityExecution(ActivityExecution execution) :
2    execution(void Execution.execute()) &&
3    target(execution);
4
5  before(ActivityExecution execution) : activityExecution(execution) {
6      // create and issue activity entry event
7  }
8
9  after(ActivityExecution execution) : activityExecution(execution) {
10     // create and issue activity exit event
11 }
12
13 pointcut actionExecution(ActionActivation activation) :
14   execution(void ActionActivation.fire(TokenList)) &&
15   target(activation);
16
17 before(ActionActivation activation) : actionExecution(activation) {
18     // create and issue activity node entry event
19 }
20
21 after(ActionActivation activation) : actionExecution(activation) {
22     // create and issue activity node exit event
23 }
24
25 pointcut extensionalValueAddedToLocus(ExtensionalValue value) :
26   call(void Locus.add(ExtensionalValue)) &&
27   args(value);
28
29 after(ExtensionalValue value) : extensionalValueAddedToLocus(value) {
30     // create and issue extensional value event of type CREATION
31 }
```

Listing 4.1: Exemplary pointcuts and advices for observing the fUML virtual machine

pletely observe the state of carried out model executions. These pointcuts and advices enable the issuance of the discussed types of events, and, thus, enable the observation of model executions being carried out by the fUML virtual machine.

Example. To illustrate the events issued during the execution of an fUML model, we make use of the example model shown in Figure 4.4. This model defines the two classes *University* and *Student*, which own the attributes *name* and *matriculationNumber*, respectively, and are associated with each other through the association *enrollment*. The class *University* defines the two operations *addNewStudent()* and *createNewStudent()*, whose behaviors are defined by the depicted activities. The operation *addNewStudent()* first calls the operation *createStudent()*, which creates a new *Student* object and initializes its *matriculationNumber* attribute, and second adds the created student to the collection of students enrolled at the university. As the operations are defined for the class *University*, the respective activities are always executed for a *University* object serving as context object of the execution. On the left-hand side of Figure 4.5, such a *University* object `o1` is shown, whose *name* attribute is set to the String value *"TU Wien"*. Furthermore, the operations require as input a String value defining the matriculation number to

be assigned to the newly created *Student* object. On the right-hand side of Figure 4.5, a valid input parameter value `pv1` is depicted, which provides the String value *"0625154"*.

Figure 4.6 and Figure 4.7 show the events issued by the event mechanism during the execution of the activity *addNewStudent* for the defined context object and input parameter value. The very first event issued is the activity entry event `e1`, which notifies about the start of the execution of the activity *addNewStudent*. The following four events report on the execution of the action *read self* and the subsequent execution of the fork node *fork university*. In particular, the start and the completion of the execution of these activity nodes are indicated by the issued activity node entry events and activity node exit events `e2` and `e3` as well as `e4` and `e5`. After the execution of the fork node *fork university*, the execution of the call operation action *call createStudent()* starts, which is indicated by the activity node entry event `e6`. The execution of this call operation action leads to the execution of the activity *createStudent* causing the issuing of the activity entry event `e7`. Next, the action *create student* contained by the activity *createStudent* is executed. Again the start and the end of executing this action are reported by a corresponding activity node entry event `e8` and activity node exit event `e10`. The execution of this create object action leads to the creation of a new *Student* object at the locus of the execution. This is reported by the extensional value event `e9`, which is created and issued before the activity node exit event `e10`. The next issued event is the activity node entry event `e11`, which indicates the start of executing the action *set matriculationNumber*. The initialization of the *matriculationNumber* attribute of the created *Student* object by this action is reported by the feature value event `e12`. Subsequently, the activity node exit event `e13` indicates the completion of the execution of this action. At this point, the execution of the activity *createStudent* is completed and therewith also the execution of the call operation action *call createStudent()* is completed. As a consequence, the activity exit event `e14` and the activity node exit event `e15` are issued by the event mechanism. In the last execution step, the action *add student* is executed. The start of executing this action is denoted by the activity node entry event `e16`, the creation of a link between the created *Student* object and the provided *University* object is indicated by the extensional value event `e17`, and the completion of executing this action is denoted by the activity node exit event `e18`. This constitutes the completion of the activity *addNewStudent*, which results in the issuing of the activity exit event `e19`.

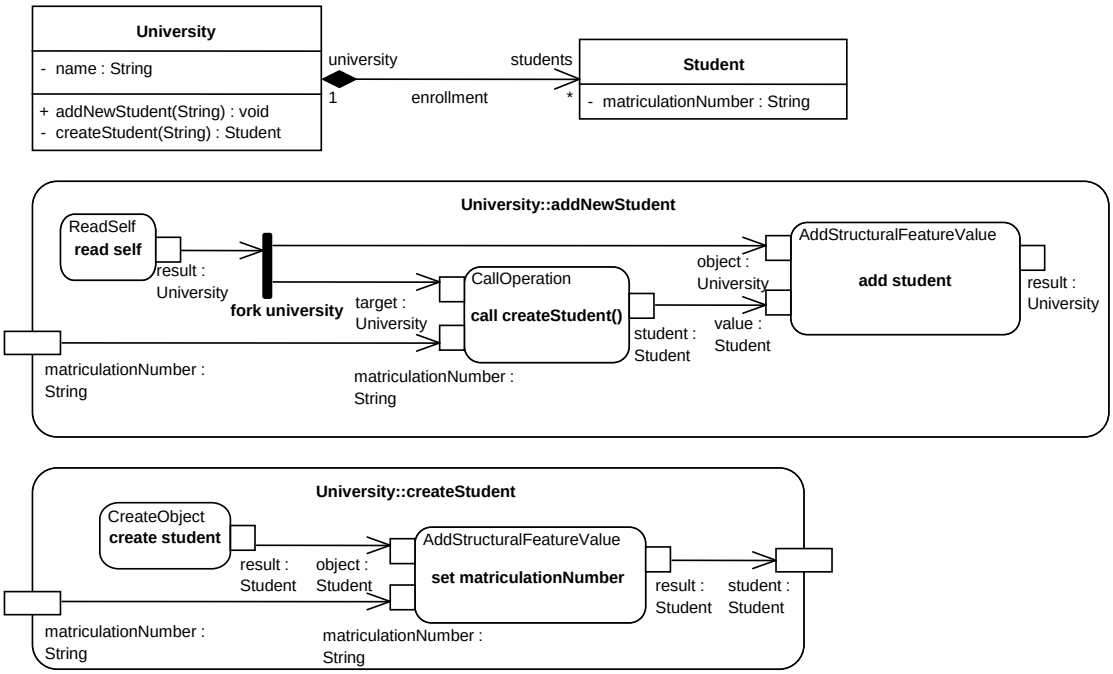


Figure 4.4: fUML execution environment extensions example: Model

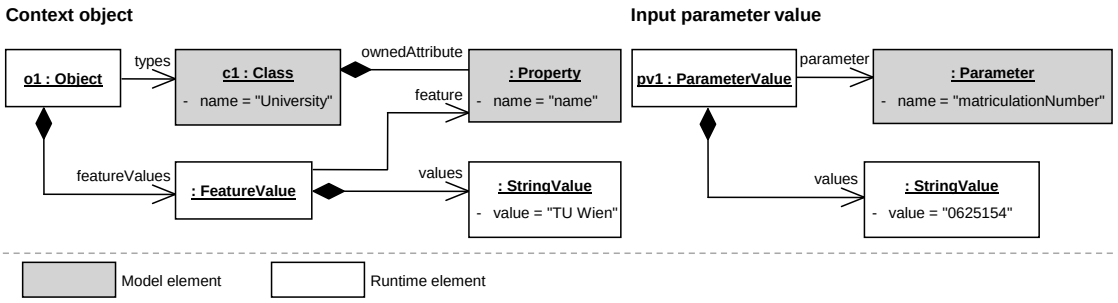


Figure 4.5: fUML execution environment extensions example: Input

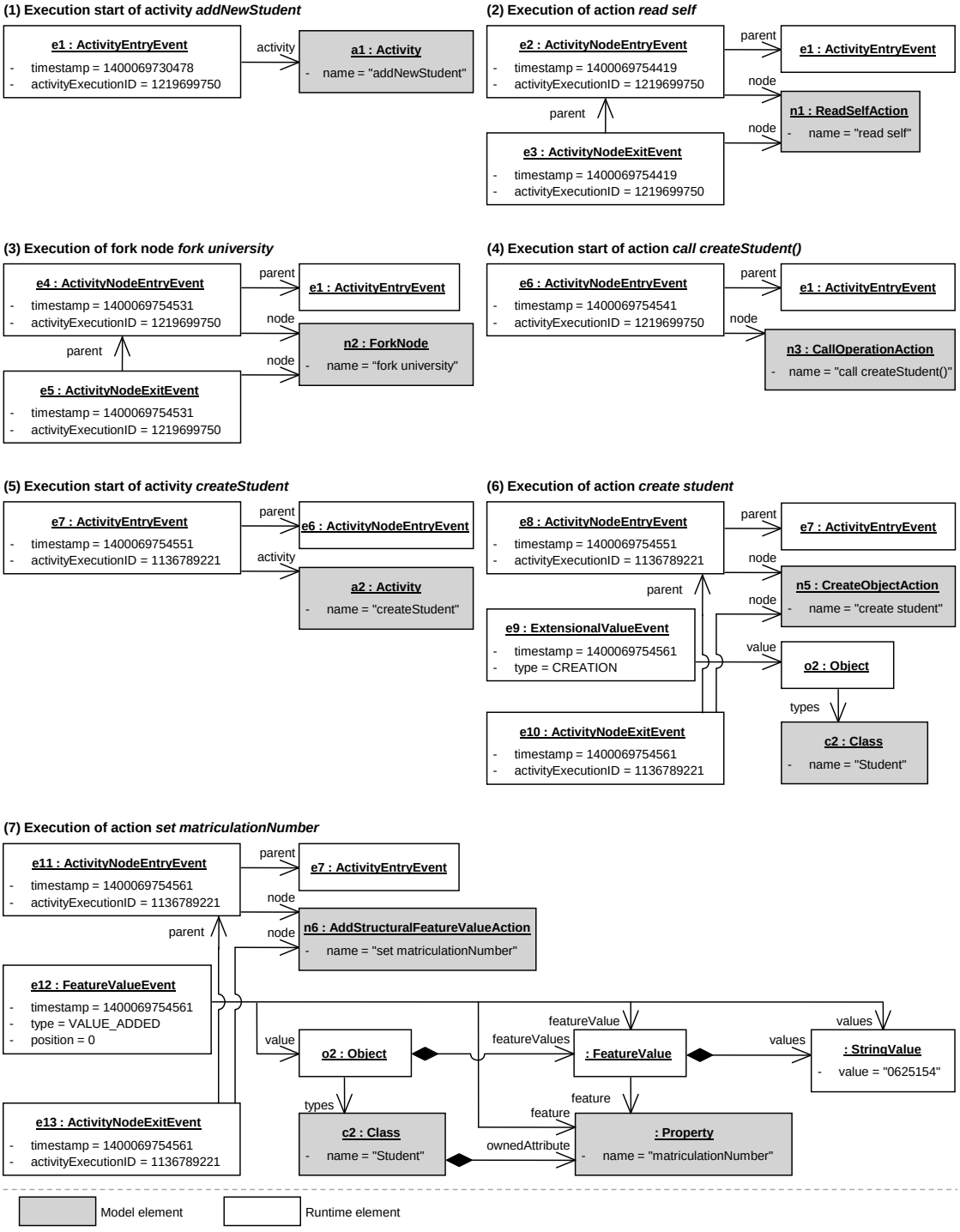


Figure 4.6: fUML execution environment extensions example: Events e1-13

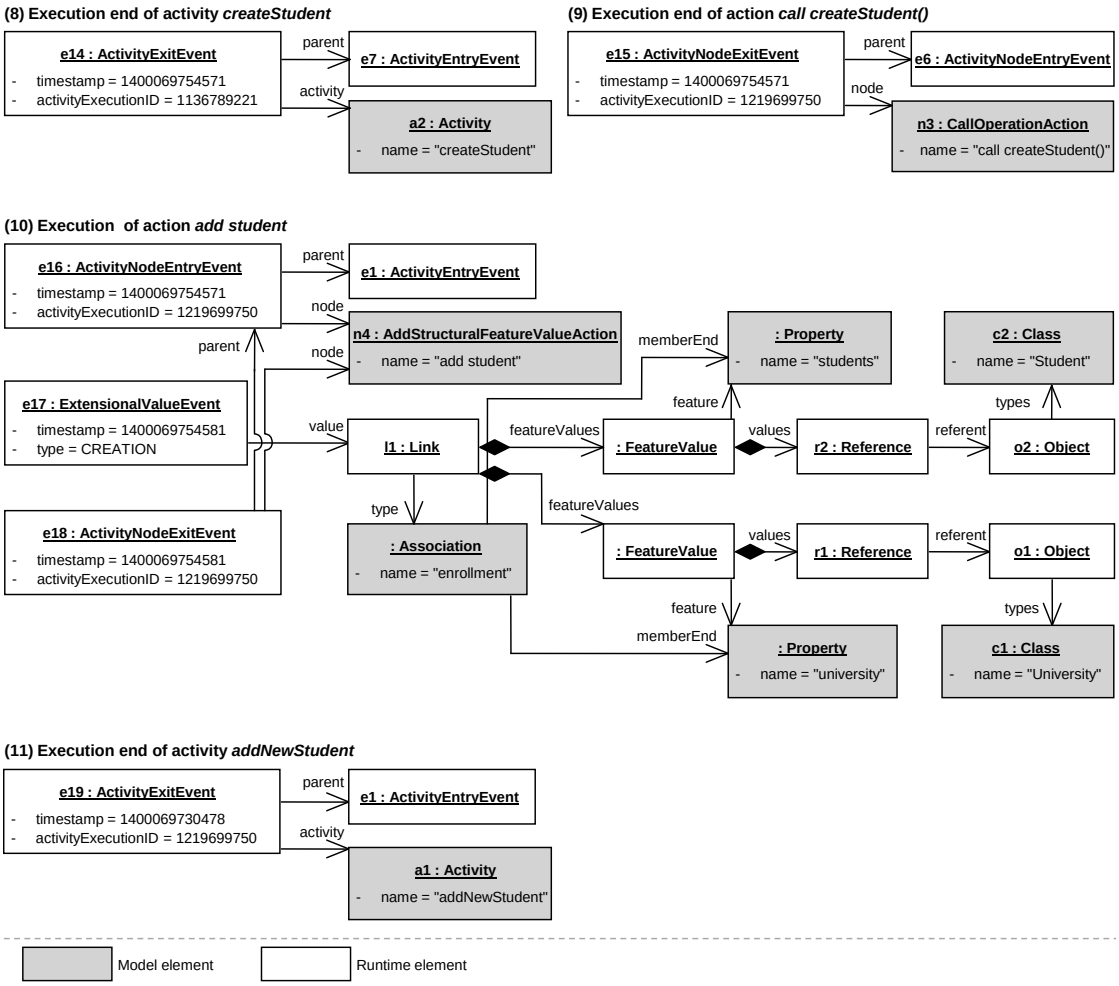


Figure 4.7: fUML execution environment extensions example: Events e14-19

4.3 Command Interface

To add *controllability* to the fUML virtual machine, we developed a *command interface* constituting the interface for utilizing the fUML virtual machine together with our developed extensions. This command interface provides the following capabilities.

- **Control of execution.** The command interface enables the control of the execution of a model being carried out by the fUML virtual machine. This includes the stepwise execution of a model, the suspension of a model execution at a particular position, and the resuming of a suspended model execution at a specific position.
- **Access to execution environment.** Furthermore, the command interface provides access to the execution environment of the fUML virtual machine. In particular, it enables the access to the locus of an ongoing model execution.
- **Observation of execution.** The command interface constitutes also the interface for observing the state of a model execution. Therefore, it provides the means to register and unregister for being notified about state changes through events issued by the introduced event mechanism.
- **Management of execution state.** The command interface records the state of model executions being carried out by the fUML virtual machine. This is required for suspending and resuming model executions at specific positions. While the recorded states are not directly exposed to users of the command interface, they are used for creating execution traces, which can be retrieved from the command interface.

Our implementation of the command interface consists of an API that provides the introduced execution control capabilities as well as an API for recording the state of ongoing model executions. To actually control model executions and record their state, we again implemented aspects for the fUML virtual machine using AspectJ. These aspects define pointcuts and advices that interrupt the fUML virtual machine after the completion of execution steps and record the current state of the model executions after each of these execution steps.

Control of execution. The main capability provided by the command interface is the ability to control the fUML virtual machine and to thereby control the execution of fUML models. The interface for making use of this capability is provided by the singleton class ExecutionContext depicted in Figure 4.8. It offers the following means for controlling model executions.

- **Start execution.** To start the execution of a model, the operations execute() and executeStepwise() are provided. In both cases, the behavior to be executed as well as the context object and input parameter values for the execution have to be passed as parameter values. If the operation execute() is used, the behavior is executed either until a breakpoint is hit during the execution or until the execution is completed. In case the operation executeStepwise() is used, the execution is suspended after the completion of each execution step. An execution step comprises either the start of the execution of a behavior, or the completion of the execution of an action or a control node contained by an executing behavior.

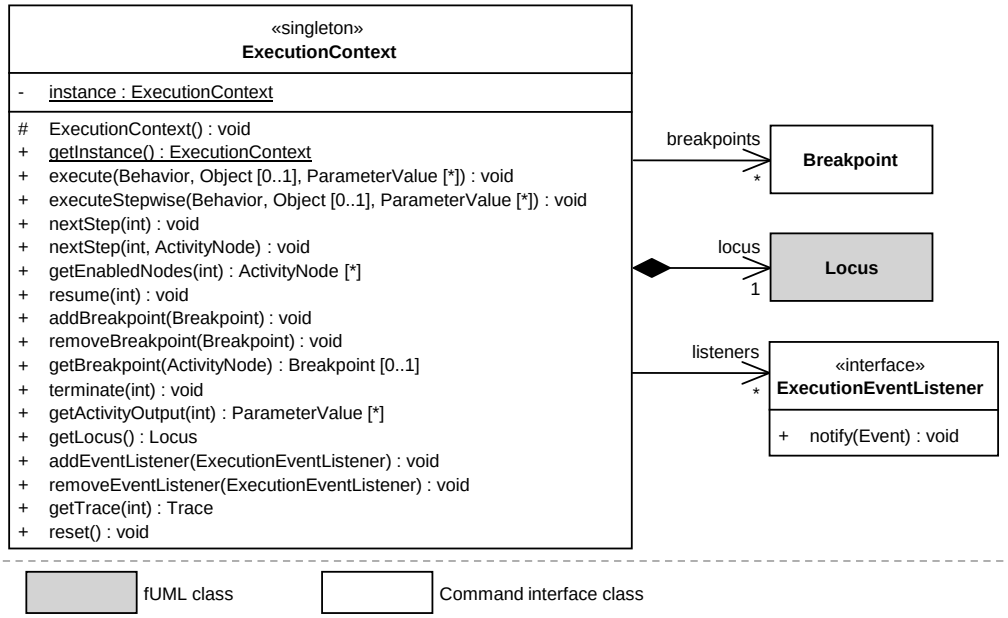


Figure 4.8: Command interface

- **Resume execution.** To resume a suspended execution, the operations nextStep() and resume() can be used. In both cases, the unique identifier of the execution to be resumed has to be passed as parameter values. This identifier is provided by the trace events issued for the respective execution. The operation nextStep() causes the execution of the next enabled action or control node of the executing activity. As multiple activity nodes can be enabled at the same time, it is possible to define a specific node to be executed in the next step. If the node is not specified, one of the enabled nodes is chosen based on a customizable node selection strategy. The enabled nodes, which can be executed in the next step, are provided by the suspension event issued after the suspension of the execution. Furthermore, the operation getEnabledNodes() can be used to retrieve the currently enabled nodes. The operation resume() continues a suspended execution until either a breakpoint is hit or the execution terminates.

- **Set breakpoints.** To add, remove, or retrieve breakpoints, the operations addBreakpoint(), removeBreakpoint(), and getBreakpoint() are provided by the command interface. Breakpoints specify that an execution should be suspended if a specific action or control node became enabled in the last step, i.e., if it can be executed in the next execution step. To remove all breakpoints, also the operation reset() can be used.

- **Terminate execution.** To terminate a suspended execution, the operation terminate() of the command interface can be used. The unique identifier of the execution to be terminated has to be provided.

- **Retrieve output.** For retrieving the output of a completed execution, the operation get-

ActivityOutput() is provided by the command interface. Again, the unique identifier of the execution whose output shall be retrieved has to be provided as parameter value to this operation.

Access to execution environment. The command interface also serves as interface to the execution environment of the fUML virtual machine. As explained in Section 3.3.4, the execution environment consists of a locus, which provides an executor as well as an execution factory. The execution factory in turn provides the primitive data types and primitive behaviors defined in the fUML Foundational Model Library, as well as implementations of strategies for semantic variation points.

The command interface provides a default configuration of the execution environment, which is used to execute models. In this default configuration, the primitive data types Boolean, Integer, String, and UnlimitedNatural, as well as all primitive behaviors for these primitive data types defined in the Foundational Model Library are available. As semantic strategies, the FIFOGetNextEventStrategy, InheritanceBasedDispatchStrategy, and FirstChoiceStrategy are configured. While the FIFOGetNextEventStrategy and the FirstChoiceStrategy constitute the default strategies for event dispatching and nondeterministic decisions defined by the fUML standard, we chose to define and use for operation dispatching our own strategy InheritanceBasedDispatchStrategy. Unlike the default dispatch strategy defined by the fUML standard, our strategy dispatches operations based on the inheritance relationships between classes, instead of based on explicitly defined redefinition relationships.

To enable the access to the execution environment, the command interface class ExecutionContext provides the operation getLocus() (cf. Figure 4.8). The primary use of this operation is to retrieve the extensional values residing at the locus of an execution or to add extensional values to the locus prior to starting an execution. Furthermore, it might be useful to add additional primitive behaviors to the execution factory of the locus. To reset the execution environment to the initial configuration, the operation reset() can be used.

Observation of execution. As explained in Section 4.2, the event mechanism introduced into the fUML virtual machine is responsible for detecting state changes of model executions and notifying about these state changes by issuing corresponding events. To register for and unregister from being notified about state changes, the command interface class ExecutionContext provides the operations addEventListener() and removeEventListener(), respectively (cf. Figure 4.8). To unregister all event listeners, the operation reset() can be used. Event listeners have to implement the interface ExecutionEventListener depicted in Figure 4.8. This interface defines the operation notify(), which is called by the event mechanism in order to issue events to the registered listeners.

Management of execution state. To enable the suspension and resumption of executions at specific positions, the command interface has to keep track of the state of the executions. Besides the information about the current position of the execution, which can be observed using the event mechanism, more detailed state information is required by the command interface. This additional required information is depicted in Figure 4.9.

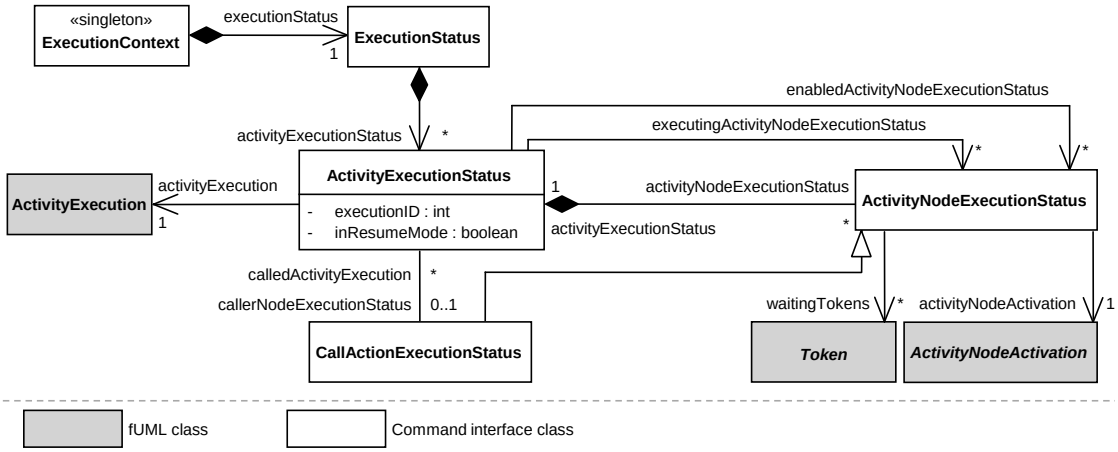


Figure 4.9: Execution state

For each ongoing activity execution, that is each activity execution started but not finished yet, the current state of the execution is captured (state class ActivityExecutionStatus), which consists of the associated activity execution (reference activityExecution), the unique identifier of the activity execution (attribute executionID), the information whether the activity execution is in resume mode or stepwise execution mode (attribute inResumeMode), the state of the call action execution which called the activity execution (reference callerNodeExecutionStatus), as well as the state of activity node executions belonging to the activity execution (reference activityNodeExecutionStatus). Furthermore, it is captured, which activity nodes are currently enabled for being executed in the next execution step (reference enabledActivityNodeExecutionStatus) and which activity nodes are currently being executed (reference executingActivityNodeExecutionStatus). For each activity node being executed in the course of an activity execution, the captured state information (state class ActivityNodeExecutionStatus) comprises the associated activity node activation (reference activityNodeActivation) and the information on which tokens have been received by the node and still have to be processed (reference waitingTokens).

Using this state information, the command interface is able to resume a suspended activity execution. In particular, the information about which activity nodes can be executed in the next execution step (reference enabledActivityNodeExecutionStatus of ActivityExecutionStatus) and which tokens have to be processed by these activity nodes (reference waitingTokens of ActivityNodeExecutionStatus), the execution can be resumed by selecting one of these enabled activity nodes and calling the operation fire() of the associated activity node activation (reference activityNodeActivation of ActivityNodeExecutionStatus) which causes the execution of this node. In case the execution of an activity terminated, i.e., no nodes of the activity are enabled after the last execution step, the information about which call action called the terminated activity execution (reference callerNodeExecutionStatus of ActivityExecutionStatus) is used to resume the activity execution to which the caller action belongs. In a similar way, the execution of called activities can be resumed based on the captured information about which activity executions resulted from

the execution of call actions (reference calledActivityExecution of CallActionExecutionStatus).

Besides this information, further information about token flows between executed activity nodes is captured, which is omitted in Figure 4.9. In particular, this includes information about tokens sent and received by activity nodes, values transported through the tokens constituting the outputs and inputs of activity nodes, and activity edges traversed by the tokens.

The state information as presented here is not directly accessible by users of the command interface, but used by the command interface itself to suspend and resume activity executions being carried out by the fUML virtual machine. Furthermore, the state information is used for creating execution traces as will be described in Section 4.4. The execution trace recorded for an execution can be obtained from the command interface using the operation getTrace() of the command interface class ExecutionContext (cf. Figure 4.8). Therefore, the unique identifier of the execution whose trace should be obtained has to be provided as parameter value.

Implementation. In order to suspend an activity execution, the command interface has to interrupt the fUML virtual machine at certain points of carrying out the execution. In particular, it has to interrupt the execution after each performed execution step, that is the completion of executing an action or control node including the sending of tokens by the executed node and the receiving of these tokens by subsequently enabled activity nodes. Hence, the fUML virtual machine has to be interrupted before the next enabled activity node is executed. This interruption of the fUML virtual machine is again realized by aspects implemented with AspectJ. Furthermore, we implemented aspects that are responsible for capturing the required state information of ongoing executions.

As an example of what these aspects look like, Listing 4.2 shows the pointcut activityNodeBecomesEnabled(), which detects calls of the operation fire() of activity node activations. As discussed in Section 3.3.2, the operation fire() is responsible for executing an activity node. Using the around advice associated with this pointcut, the start of the execution of the activity node is prevented and the activity node is instead added to the execution state of the command interface as an enabled node.

To resume a suspended activity execution, one of the currently enabled nodes of this activity execution has to be executed. Listing 4.3 shows how this is done by the operation nextStep() of the command interface class ExecutionContext. First, the state of the activity execution that shall be resumed is retrieved and used to obtain the state of the enabled activity node that shall be executed in the next execution step. The state of this enabled activity node comprises the associated activity node activation and the tokens to be processed by the activity node execution. Using this information, the node can be executed by calling the operation fire() of the respective activity node activation passing the tokens to be processed.

Example. Figure 4.10 illustrates the usage of the command interface for executing the example model depicted in Figure 4.4 for the input depicted in Figure 4.5 and controlling this execution. The instance `context` of the class ExecutionContext represents the command interface, which is responsible for executing the fUML model as well as for providing the means for observing and controlling it. The instance `listener` of the interface ExecutionEventListener uses the command interface to execute the fUML model and observe as well as control the execution.

```
1  pointcut activityNodeBecomesEnabled(ActivityNodeActivation activation, TokenList tokens) :
2    call(void ActivityNodeActivation.fire(TokenList)) &&
3    withincode(void ActivityNodeActivation.receiveOffer()) &&
4    target(activation) &&
5    args(tokens);
6
7  void around(ActivityNodeActivation activation, TokenList tokens) :
8    activityNodeBecomesEnabled(activation, tokens) {
9      if (activation instanceof ObjectNodeActivation) {
10       proceed(activation, tokens);
11     } else {
12       ActivityExecution currentExecution = activation.getActivityExecution();
13       ActivityExecutionStatus exestatus = ExecutionContext.getInstance().executionStatus.
             ↪getActivityExecutionStatus(currentExecution);
14       exestatus.addEnabledActivityNodeExecutionStatus(activation, tokens);
15     }
16   }
```

Listing 4.2: Exemplary pointcut and advice for controlling the fUML virtual machine

```
1  void nextStep(int executionID, ActivityNode node) {
2    ActivityExecutionStatus activityExecutionStatus = executionStatus.getActivityExecutionStatus(
         ↪executionID);
3    ActivityNodeExecutionStatus nodeExecutionStatus = activityExecutionStatus.
         ↪getEnabledActivityNodeExecutionStatus(node);
4    activityExecutionStatus.addExecutingActivityNodeExecutionStatus(nodeExecutionStatus);
5    ActivityNodeActivation activation = nodeExecutionStatus.getActivation();
6    TokenList tokens = nodeExecutionStatus.getTokens();
7    activation.fire(tokens);
8  }
```

Listing 4.3: Exemplary command interface operation for controlling the fUML virtual machine

For observing the execution, i.e., for being notified about state changes of the execution through events, the object `listener` first registers itself at the command interface as event listener by calling the operation addEventListener(). Thereafter, it starts the execution of the activity *addNewStudent* by calling the command interface operation executeStepwise(). Consequently, the command interface starts the execution of the activity by passing the activity to the fUML virtual machine and notifies the listener about this by issuing the activity entry event `e1` (cf. Figure 4.6) using the operation notify(). After the fUML virtual machine has started the execution of the activity and determined the initially enabled nodes, the command interface suspends the execution by interrupting the fUML virtual machine and issues the suspend event `se1` informing about the suspension. In addition, the suspend event carries the information that the execution was suspended after starting the activity *addNewStudent* and that the action *read self* is enabled and can be executed in the next execution step. To resume the execution, the `listener` object calls the command interface operation nextStep() passing the unique identifier of the execution,

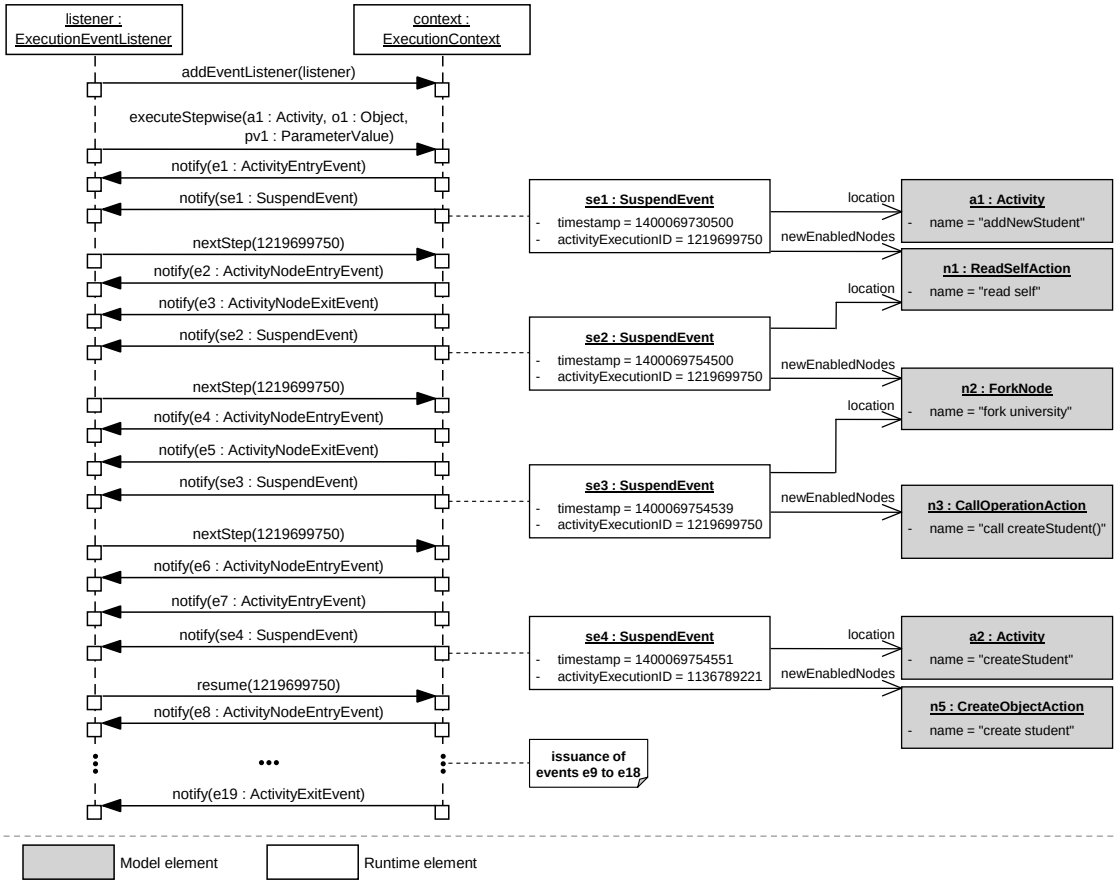


Figure 4.10: fUML execution environment extensions example: Commands

which was transfered to the `listener` by the previously received trace events. The call of the operation nextStep() causes the execution of the action *read self* and the issuing of the activity node entry event `e2` and the activity node exit event `e3` (cf. Figure 4.6). After finishing the execution of this action, the whole execution is again suspended and the suspend event `se2` is issued. By calling the operation nextStep(), the execution is resumed resulting in the execution of the fork node *fork university*, which became enabled in the last execution step. Again, an activity node entry event `e4` and an activity node exit event `e5` (cf. Figure 4.6), as well as a suspend event `se3` are issued. In the following execution step, the action *call createStudent()* starts executing which results in the start of the execution of the activity *createStudent*. As a consequence, the activity node entry event `e6` and the activity entry event `e7` (cf. Figure 4.6), as well as a suspend event `se4` are issued to the `listener` object. As a last command, the `listener` resumes the execution using the command interface operation resume(). This leads to the execution of the actions *create student*, *set matriculationNumber*, and *add student*, as well as to the issuance of the events `e9` to `e18` as depicted in Figure 4.6 and Figure 4.7. The activity

exit event `e19` constitutes the last event issued for the execution of the activity *addNewStudent* and informs about the completion of this execution.

4.4 Trace Model

Our third extension of the fUML execution environment is concerned with analyzability, in particular *dynamic analyzability*. As described in Section 2.1.3, dynamic analyzability is the ability to analyze properties of a running program—in our case properties of a running fUML model. It is typically done by analyzing *execution traces* providing an abstract representation of the runtime behavior of the executed program. Various execution trace formats have been proposed for object-oriented and procedural programming languages [60, 61]. However, those formats focus on traditional programming languages and, hence, provide runtime concepts tailored to these languages, such as routine calls, which are not directly transferable to runtime concepts of fUML models. Moreover, essential runtime concepts of fUML models, such as token flows, cannot be represented with existing trace formats.

To capture execution traces of fUML models adequately and precisely and, hence, provide the basis for performing dynamic analyses of fUML models, we developed a dedicated execution trace format in the form of a metamodel. Using this metamodel, *trace models* can be created constituting execution traces, which capture the runtime behavior of fUML models. In particular, the following runtime information about the execution of fUML models is captured in those trace models.

- **Executions.** Trace models of fUML model executions contain information about which model elements, in particular which activities, actions, and control nodes, have been executed. Additionally the chronological execution order of model elements is captured.
- **Inputs and outputs.** Another important information captured by trace models is the input processed by model elements as well as the output produced by them.
- **Token flows.** Besides the input and output of model elements, also token flows between the model elements through activity edges for passing either values or control is captured by trace models. This enables reasoning about input/output relations between model elements.

The trace model of an fUML model execution is instantiated as soon as the execution starts and it is continuously updated after each execution step. Thus, the trace model is available also for partially executed models and captures the runtime behavior of the model until the respective point of execution. Therefore, the event mechanism and the command interface are used to keep track of the execution progress and to retrieve the runtime information to be captured in the trace model, respectively.

Executions. To perform a fine-grained analysis of the runtime behavior of an fUML model, knowledge about which parts of the model have been executed is required. Figure 4.11 shows the excerpt of the trace metamodel, which enables capturing this knowledge.

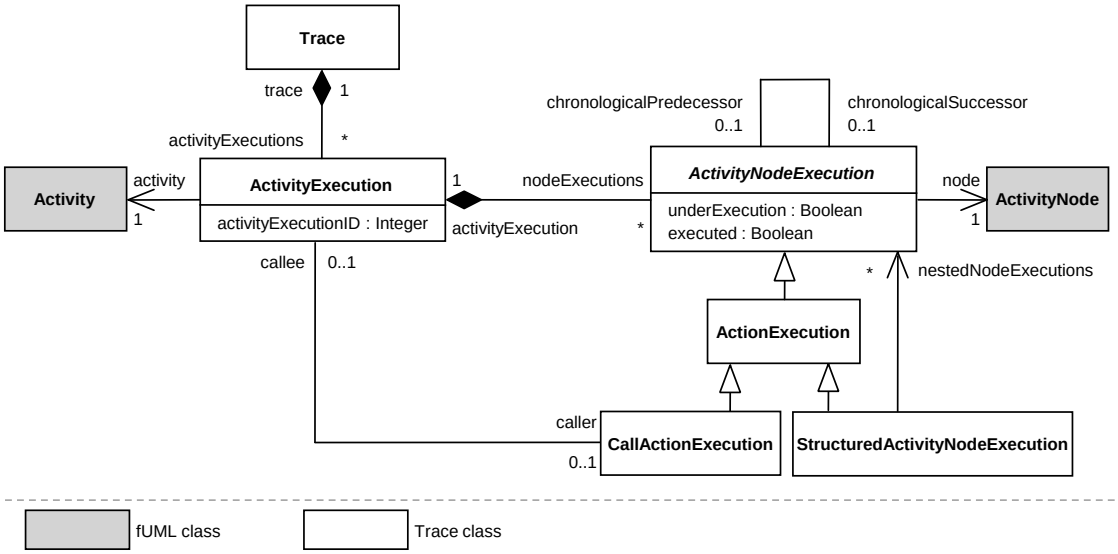


Figure 4.11: Excerpt of trace metamodel for capturing executions

A trace model of an fUML model (metaclass Trace) captures the information about which activities and activity nodes have been executed, the chronological execution order of activities and activity nodes, the call hierarchy among executed activities, as well as the nesting of activity node executions.

- **Activity executions.** The trace model of an fUML model captures information about the execution of activities (metaclass ActivityExecution) carried out during the execution of the model. This includes the activity constituting the starting point of the model execution, that is the activity provided to the command interface operation execute() or executeStepwise(), as well as all activities called in the course of executing this starting point activity through the execution of call actions.

- **Activity node executions.** For each executed activity, it is recorded which activity nodes have been executed (metaclass ActivityNodeExecution). In particular, it is recorded which actions and control nodes have been executed. Again, like for the events issued for the execution of activity nodes, the execution of object nodes is not recorded explicitly in the trace, because they are always executed in the course of executing the associated action or activity (cf. Section 4.2).

  As mentioned before, the trace model of an fUML model is updated during the execution of the model such that the trace model always captures the runtime behavior of model until the current execution position. Besides the capturing of already executed activity nodes, the trace also captures which activity nodes are currently being executed, as well as which activity nodes are currently enabled for being executed in the next execution step. The Boolean attributes underExecution and executed enable the determination of

the state of an activity node execution. Completed activity node executions have these attributes set to the values underExecution = false, executed = true, ongoing activity node executions to underExecution = true, executed = false, and enabled activity node executions to underExecution = false, executed = false.

Please note that if the same activity node is executed multiple times in the course of the same activity execution, multiple activity node executions are captured for this activity node. The same is true for the execution of activities. If the same activity is executed multiple times in the course of the execution of an fUML model, multiple activity executions for this activity are recorded in the trace model.

- **Chronological execution order.** For all executed activity nodes, the chronological order in which they have been executed is captured (references chronologicalPredecessor and chronologicalSuccessor of ActivityNodeExecution). This chronological order is global for the execution of the model, instead of local to the respective activity execution. Through the chronological ordering of call action executions captured in the trace (metaclass CallActionExecution), and the associations of these call action executions with the triggered activity execution (reference callee), also the chronological order and interleaving of activity executions is captured.

- **Call hierarchy among activity executions.** Activity executions can cause the execution of other activities through the execution of call actions. The resulting call hierarchy among activity executions is explicitly captured in the trace model (reference caller of ActivityExecution). This information can be used to analyze interactions between activities as well as interactions between objects, as activities are executed for dedicated objects if they define the behavior of class operations.

- **Nesting of activity node executions.** For the execution of structured activity nodes (metaclass StructuredActivityNodeExecution), it is recorded which activity nodes grouped by the structured activity node have been executed (reference nestedNodeExecutions). This information is required to associate activity node executions to structured activity node executions, as the same activity nodes and structured activity nodes might be executed multiple times.

Inputs and outputs. To enable the dynamic analysis of the relations between the inputs, outputs, and behavior of a model, the inputs processed as well as outputs produced by executed activities and activity nodes are captured in the trace model. Figure 4.12 shows the excerpt of the trace metamodel capturing information about inputs and outputs.

- **Values.** As in any object-oriented language, also in fUML the behavior of a system is defined in terms of object manipulations, which are expressed using dedicated actions for creating and destroying objects, modifying attribute values of objects, as well as creating and destroying links between objects. Object manipulations performed during the execution of an fUML model are captured in the trace model. Therefore, each manipulated value is captured (metaclass ValueInstance). In particular, objects and links existing at

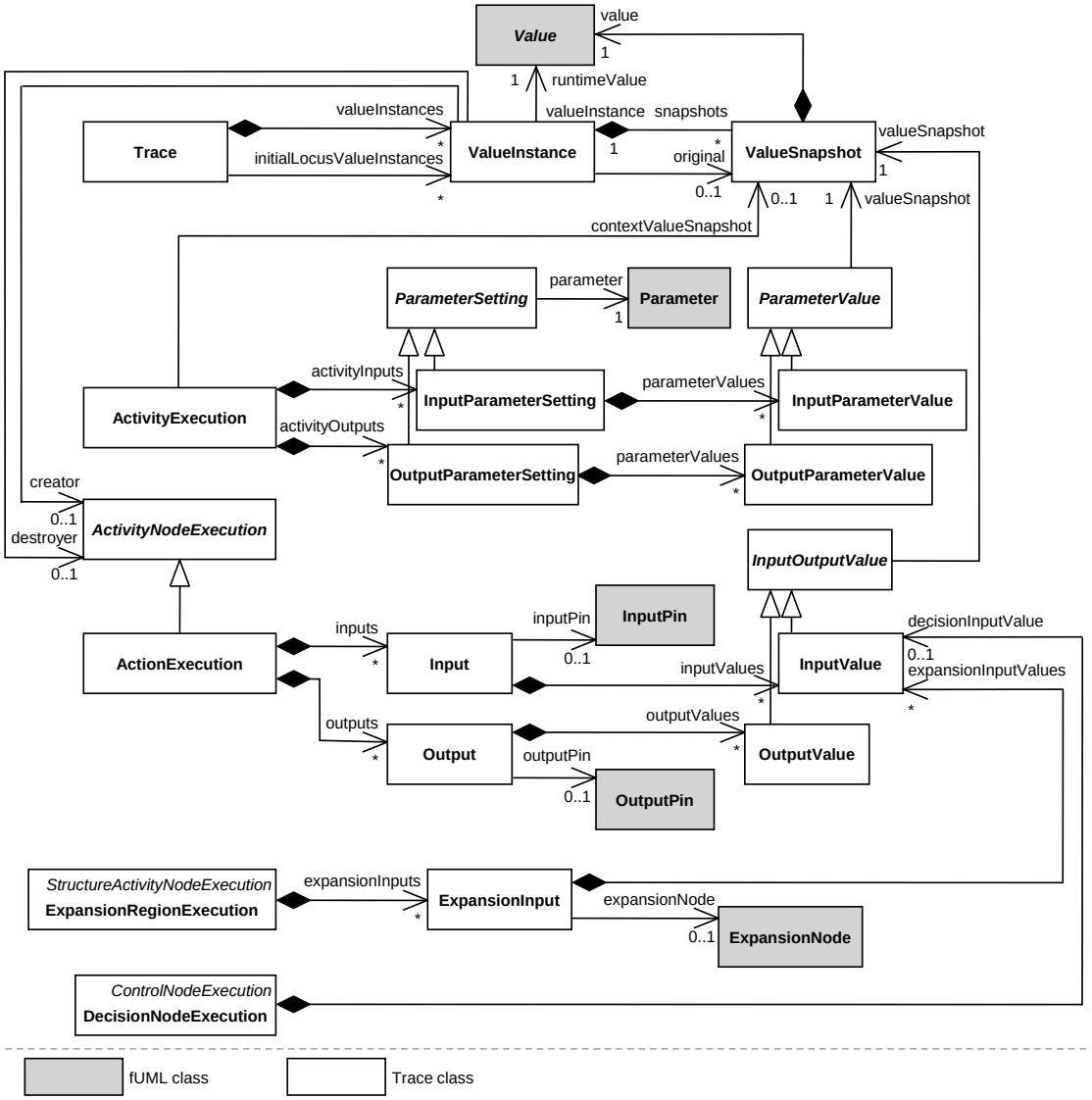


Figure 4.12: Excerpt of trace metamodel for capturing inputs and outputs

the locus prior to the model execution are captured (reference initialLocusValueInstances of Trace), as well as values created during the model execution. Each manipulation of a value is recorded by capturing a snapshot of the manipulated value (metaclass ValueSnapshot), which constitutes a deep copy of the respective value after the manipulation took place (reference value). For values created or destroyed during the model execution, the action execution causing this creation or destruction is captured (references creator and destroyer of ValueInstance). Therewith, the state of all objects at any point in time of a model execution is captured by the trace model.

- **Activity inputs and outputs.** For activity executions, the trace model captures the processed input values as well as the provided output values (metaclasses InputParameterSetting and InputParameterValue, as well as OutputParameterSetting and OutputParameterValue). In particular, it captures which snapshots of values were received as input and provided as output (reference valueSnapshot of ParameterValue) by an activity execution through which input and output parameter, respectively (reference parameter of ParameterSetting).

  Besides input and output values of activity executions, the trace model also captures the context object of the execution, in particular, the snapshot of the context object at the time when the execution started (reference contextValueSnapshot of ActivityExecution).

- **Action inputs and outputs.** The inputs and outputs of action executions are captured in a similar way as the inputs and outputs of activity executions (metaclasses Input and InputValue respectively Output and OutputValue). In particular, it is captured which snapshots of values were received as input and provided as output (reference valueSnapshot of InputOutputValue) by an action execution through which input pin and output pin, respectively (references inputPin of Input and outputPin of Output).

- **Expansion region inputs.** Expansion regions are a special type of action, which can be used to process collections of values. The collections of values to be processed are received by an expansion region through expansion nodes, which are like input pins a special type of object node. In the trace model, the collection of values received by an expansion region execution as input is captured separately from values received through input pins (metaclass ExpansionInput).

- **Decision node inputs.** Control nodes can be used to route values between actions and do not process and produce values on their own. However, decision nodes constitute an exception, as they process values provided through decision input flow edges in order to evaluate the guard conditions of outgoing edges. Hence, these processed values are captured in the trace (reference decisionInputValue of DecisionNodeExecution).

Token flows. In order reason about dependencies between activity nodes, data dependencies resulting from object flows and control dependencies resulting from control flows have to be analyzed. Figure 4.13 shows the part of the trace metamodel, which is used to capture object flows and control flows among executed activity nodes.

All tokens flowing through an activity during execution are captured in the trace model (metaclass TokenInstance) together with the information about which edges have been traversed by the respective token (reference traversedEdges). Thereby, it is distinguished between object tokens representing object flows (metaclass ObjectTokenInstance) and control tokens representing control flows (metaclass ControlTokenInstance).

- **Object flows.** During the execution of an activity, object tokens (metaclass ObjectTokenInstance) are transferred between activity nodes via activity edges in order to pass values between the activity nodes (reference transportedValue). Object tokens are created for

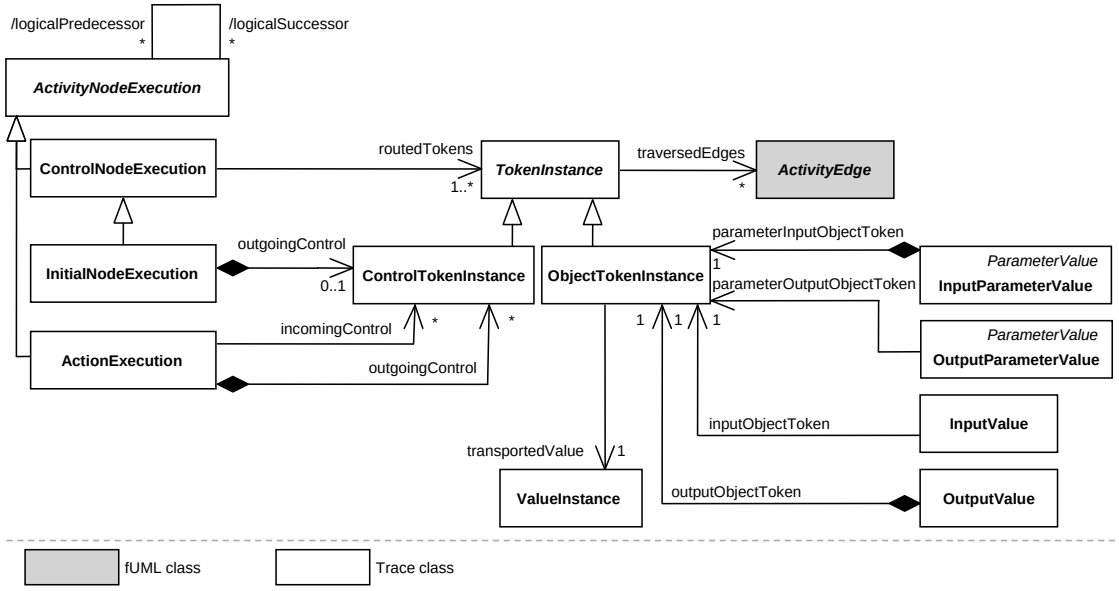


Figure 4.13: Excerpt of trace metamodel for capturing token flows

each input parameter value of the activity (reference parameterInputObjectToken of InputParameterValue) as well as for each output value of executed actions (reference outputObjectToken of OutputValue). Created object tokens can be passed to output activity parameter nodes for transporting the output parameter values of the activity (reference parameterOutputObjectToken of OutputParameterValue), to input pins of actions, expansion nodes of expansion regions, or decision nodes for transporting input values (reference inputObjectToken of InputValue), or to control nodes for further routing the transported values through the activity (reference routedTokens of ControlNodeExecution).

- **Control flows.** The trace model also captures the flow of control tokens between activity nodes (metaclass ControlTokenInstance), which cause control dependencies among activity nodes. Thereby, control tokens are created either by the execution of initial nodes or by the execution of actions (references outgoingControl of InitialNodeExecution and ActionExecution). Created control tokens are either passed to control nodes, which route the tokens further through the activity (reference routedTokens of ControlNodeExecution), or to actions (reference incomingControl of ActionExecution).

- **Dependencies.** From the token flow information captured in the trace model, the dependencies between executed activity nodes can be derived. In particular, the logical predecessors and successors of an activity node can be derived. An executed activity node `A` is the logical predecessor of another executed activity node `B`, if `A` provided object tokens or control tokens to `B` (reference logicalPredecessor of ActivityNodeExecution). Inversely, an executed activity node `B` is the logical successor of another executed activity node `A`, if

B received object tokens or control tokens from A (reference logicalSuccessor of ActivityNodeExecution).

Implementation. For creating trace models of fUML model executions, the state of the executions captured by the command interface as described in Section 4.3 (cf. Figure 4.9) is used. The captured execution state provides all necessary information required about the runtime behavior of executed models to produce trace models according to the format defined by the trace metamodel. The trace is created as soon as the execution of the first activity contained by an executed fUML model starts and it is updated on each state change of the execution. To observe the state changes of model executions, the event mechanism introduced in Section 4.2 is used. The created trace models are updated according to the issued events as described in the following.

- **Activity entry event.** When an activity execution is started, it is added to the respective trace model and its caller action execution, context object, and input parameter values are accordingly assigned.

- **Suspend event.** When an execution is suspended, the nodes, which became enabled in the last execution step, are added to the trace model. Furthermore, the inputs of the enabled nodes as well as tokens received by them are accordingly assigned.

- **Activity node entry event.** When the execution of an activity node starts, the respective execution captured in the trace model is marked as being under execution and the reference to its chronological predecessor is accordingly assigned. Furthermore, if the started activity node execution is an action execution or decision node execution, its input values are updated according to the latest captured value snapshots.

- **Activity node exit event.** When the execution of an activity node ends, the respective execution captured in the trace model is marked as being executed and sent tokens are accordingly assigned. Furthermore, if the ended activity node execution is an action execution, its output values are recorded.

- **Activity exit event.** When the execution of an activity ends, its output parameter values are recorded in the trace model.

- **Extensional value event.** When a new value is created, which is indicated by an extensional value event of the type CREATION, a new value instance is recorded for this value, a first value snapshot is produced for this value and added to the newly recorded value instance, and the creator of the value instance is accordingly set. When an existing value is destroyed, which is indicated by an extensional value event of the type DESTRUCTION, the destroyer of the respective value instance is accordingly set. When an existing value is modified, which is indicated by an extensional value event of a type other than CREATION or DESTRUCTION, a new value snapshot is produced for this value and added to the respective value instance.

Because the trace model is continuously updated during the execution of an fUML model, using the command interface operation getTrace() (cf. Figure 4.8), an up to date trace can be obtained at any point in time of the execution.

Example. Figure 4.14, Figure 4.15, and Figure 4.16 depict excerpts of the trace model captured for the execution of our example fUML model (cf. Figure 4.4 and Figure 4.5).

The trace model excerpt depicted in Figure 4.14 shows which parts of the fUML model have been executed, namely the activities *addNewStudent* (`a1exe`) and *createStudent* (`a2exe`), as well as their contained activity nodes in the following chronological order: *read self* (`n1exe`), *fork university* (`n2exe`), *call createStudent()* (`n3exe`), *create student* (`n5exe`), *set matriculationNumber* (`n6exe`), and *add student* (`n4exe`). Furthermore, it is shown that the execution of the call operation action *call createStudent()* belonging to the execution of the activity *addNewStudent* caused the execution of the activity *createStudent* (references caller and callee between `n3exe` and `a2exe`).

In Figure 4.15, the excerpt of the trace model is depicted, which captures the inputs and outputs of the execution of the activity *createStudent* (`a2exe`) as well as the inputs of outputs of its contained executed actions (`n5exe` and `n6exe`). The input of the activity execution was the String value *"0625154"* `v1` captured by the value snapshot `vs1` of the value instance `vi1` (cf. input parameter setting `ips1` and input parameter value `ipv1` of the activity execution `a2exe`). The output of the activity execution was the object `v3` of type *Student* with the attribute *matriculationNumber* set to the String value *"0625154"* captured by the value snapshot `vs3` of the value instance `vi2` (cf. output parameter setting `ops1` and the output parameter value `opv1` of the activity execution `a2exe`). The execution of the action *create student* (`n5exe`) provided as output the new instantiated *Student* object `v2` captured by the value snapshot `vs2` of the value instance `vi2` (cf. output `o1` and output value `ov1` of the action execution `n5exe`). The inputs provided to the execution of the action *set matriculationNumber* (`n6exe`) were this new instantiated *Student* object `v2` and the String value `v1` captured by the already mentioned value snapshots `vs2` and `vs1` (cf. inputs `i1` and `i2` as well as input values `iv1` and `iv2`). The produced output was the *Student* object `v3` with set *matriculationNumber* attribute captured by the snapshot `vs3` (cf. output `o2` and output value `ov2`). Please note that the two *Student* objects `v2` and `v3` constitute two distinct snapshots (`vs2` and `vs3`) of the same runtime object. The first snapshot captures the state of this object after its instantiation and the second snapshot after the initialization of the *matriculationNumber* attribute.

Figure 4.16 shows the excerpt of the trace model, which captures the token flow between the actions *create student* (`n5exe`) and *set matriculationNumber* (`n6exe`). Between these actions one object token `t1` was exchanged, which transported the *Student* object represented by the value instance `vi2` (cf. Figure 4.15). Hence, the execution of the action *create student* is the logical predecessor of the execution of the action *set matriculationNumber* (references logicalPredecessor and logicalSuccessor between `n5exe` and `n6exe`).

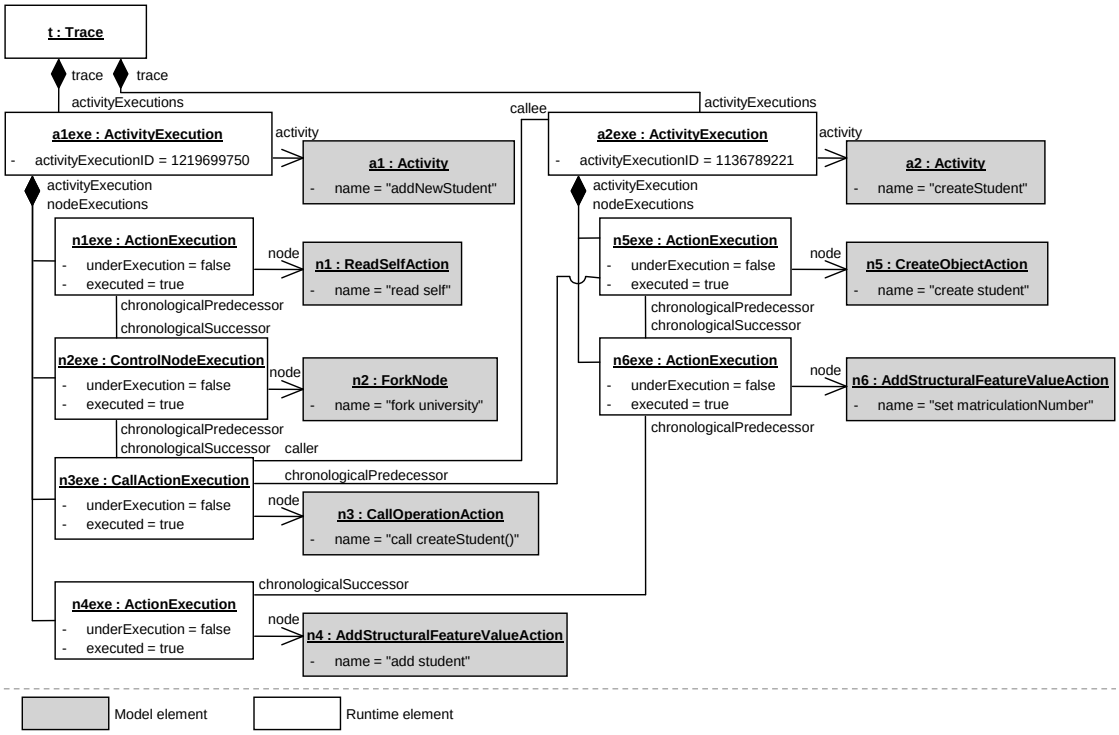


Figure 4.14: fUML execution environment extensions example: Trace model excerpt capturing executions

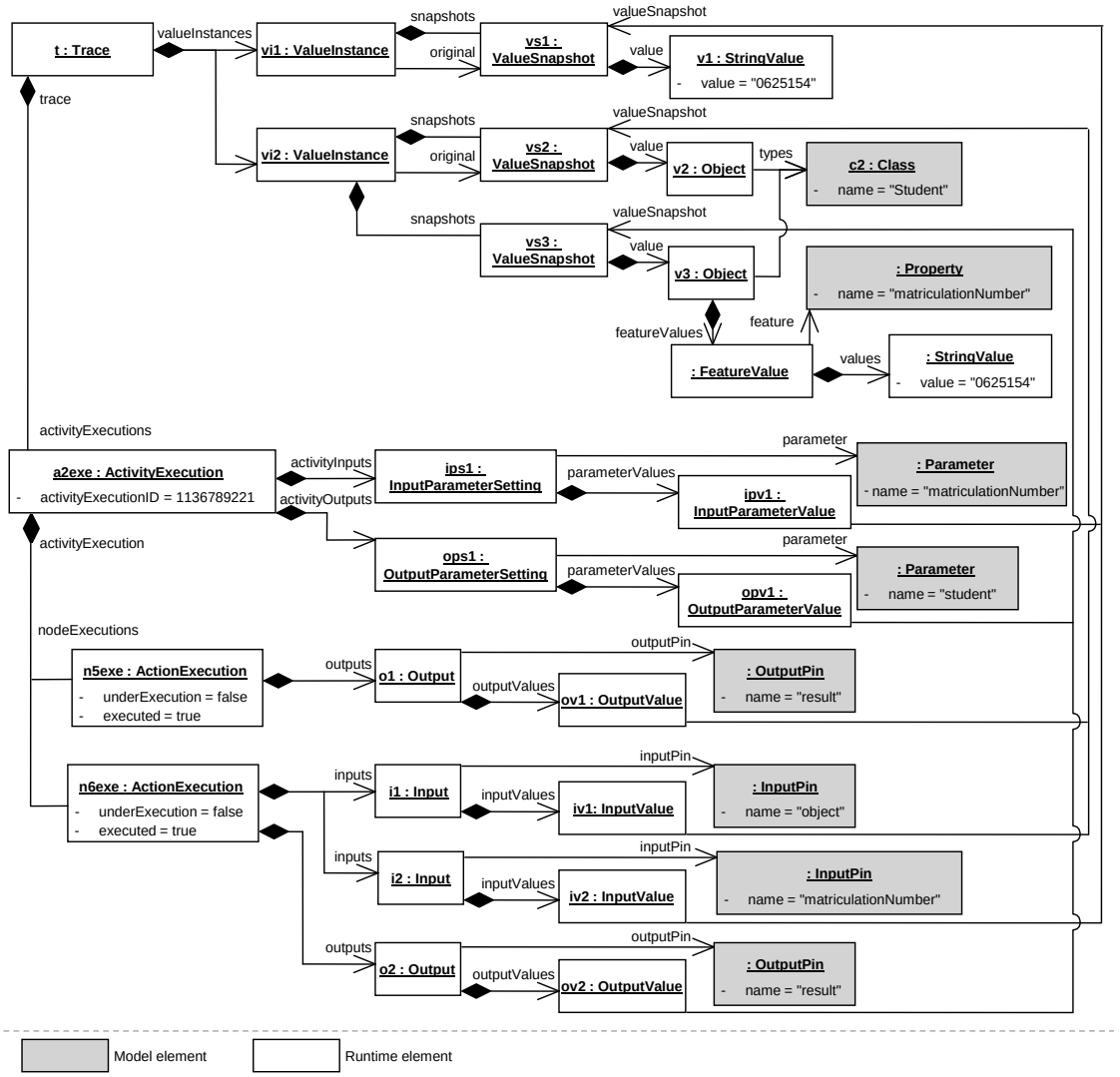


Figure 4.15: fUML execution environment extensions example: Trace model excerpt capturing inputs and outputs

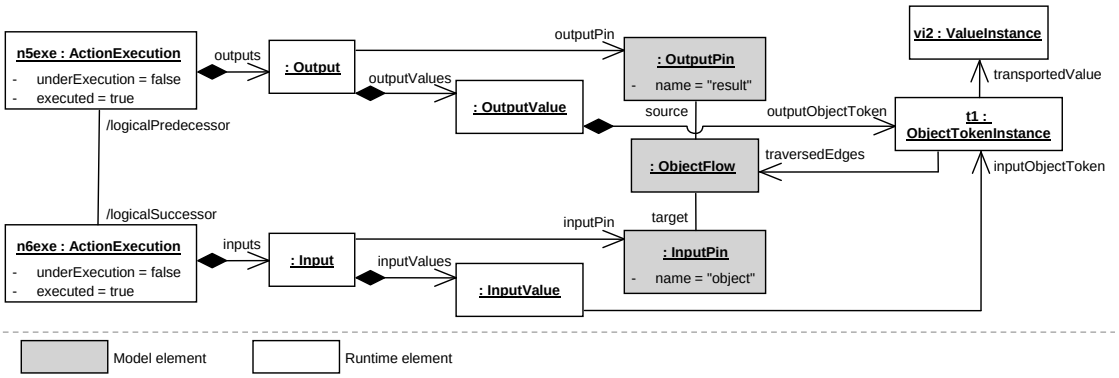


Figure 4.16: fUML execution environment extensions example: Trace model excerpt capturing token flows

4.5 Summary

We introduced the characteristics observability, controllability, and analyzability into the fUML virtual machine in order to enable the development of analysis methods for fUML. Therefore, we extended the execution environment of fUML with an event mechanism enabling runtime observation, a command interface enabling execution control, and a trace model enabling runtime analysis. These extensions are summarized in the following.

Event mechanism. The event mechanism enables the runtime observation of model executions being carried out by the fUML virtual machine. Therefore, it detects state changes of ongoing model executions and notifies about these state changes by issuing events. In particular, events reporting on the current position of the execution, as well as on modifications of extensional values residing at the locus of the execution are issued.

Command interface. The command interface provides execution control over model executions being carried out by the fUML virtual machine. In particular, it enables the stepwise execution of models, the suspension of the execution of a model at a particular position, and the resumption of the execution after suspension. Furthermore, the command interface constitutes the interface for utilizing the fUML virtual machine as well as the extensions. Thus, it provides the means for starting the execution of a model (interface to fUML virtual machine), accessing the execution environment of fUML (interface to fUML execution environment), registering and unregistering for being notified about state changes of executions through events (interface to event mechanism), and retrieving execution traces (interface to trace model).

Trace model. To enable the conduct of dynamic analyses of fUML models, trace models are recorded for model executions carried out by the fUML virtual machine. Therefore, we developed a trace metamodel, which is designed specifically to capture the runtime behavior of

UML activities. In particular, a trace model provides information about executed model elements and their chronological execution order, inputs and outputs of executed model elements, as well as token flows and input/output relations between executed model elements. Thereby, the trace model of a model execution is continuously updated during execution, such that the runtime behavior of the model until the respective point of the execution is reflected and can be analyzed.

The introduced extensions enable the development of important analysis methods for fUML based on the fUML virtual machine for supporting the validation, verification, and comprehension of fUML models. We evaluated this potential of our extensions of the fUML execution environment by implementing dedicated analysis tools based on these proposed extensions. In particular, we developed a debugger, a testing framework, as well as a performance analysis tool for fUML. We will report on these developed analysis tools in Section 7.1.2.

Table 4.1 summarizes which extension constitutes the main basis for the development of which analysis methods. While the event mechanism as well as the command interface constitute the main prerequisite for developing debugging methods, the trace model builds the crucial basis for developing testing, dynamic analysis, and non-functional property analysis methods. However, please note that the extensions also provide important utilities for developing analysis methods other than the ones indicated in Table 4.1. As an example, the information captured in the trace model can be leveraged for implementing debuggers to display, for instance, the call hierarchy leading to the last executed model element and the evolution of the context object of the currently executing activity. Similarly, the command interface can be used by non-functional property analysis tools to control the execution order of activity nodes.

As formal analysis is not considered by this thesis, it is omitted in Table 4.1. However, as pointed out by Romero *et al.* [52], the semantics specification of bUML with the first-order logic formalism PSL could be utilized to perform formal analysis through theorem proving. Furthermore, a translational semantics approach mapping fUML to a semantic domain suitable for formal analysis could be applied. In Section 2.3.1, we gave an overview of existing work applying this approach to different versions and subsets of UML activities. The work of Abdelhalim *et al.* [1,2] is an example of applying this approach to fUML by translating fUML models

	Event mechanism	**Command interface**	**Trace model**
Testing			X
Dynamic analysis			X
Debugging	X	X	
Non-functional property analysis			X

Table 4.1: Possible applications of fUML execution environment extensions for implementing model analysis methods

into CSP for verifying model consistency and deadlock freeness through model checking.

Besides evaluating the adequacy of our developed extensions for the development of analysis methods for fUML, we also evaluated the performance overhead caused by the extensions. The evaluation of the extensions is discussed in Section 7.1.

4.6 Related Work

We are aware of four extensions of fUML's execution environment, which are related to our work. An overview of these extensions is provided in Table 4.2

Laurent *et al.* [88] developed extensions of fUML regarding execution control and runtime observation for enabling the debugging of fUML models. Their extensions are very similar to our command interface and event mechanism and might have been developed in parallel to our extensions as they have been presented only five months after our work. They introduce a dedicated controller into the execution model of fUML, which enables starting, stopping, pausing, and resuming the execution of an fUML model, as well as setting breakpoints, conditional breakpoints, and watchpoints. Furthermore, an ongoing execution can be observed by listening to events issued by the controller concerning the entry and exit of activities and activity nodes, the modification of extensional values, and the sending of signals. Like our implementation, the implementation of Laurent *et al.* introduces the proposed extensions into the reference implementation of the fUML virtual machine using aspect-oriented programming. Furthermore, they developed an fUML debugger integrated with Eclipse based on the obtained extended fUML virtual machine.

Tatibouet *et al.* [157] propose an extension of fUML enabling the control of the execution order of activity nodes during the execution of an fUML model. The aim of their extension is to provide a simulation framework based on fUML, which supports arbitrary models of computation defining the execution order of activity nodes, as well as non-functional aspects, such as time. Therefore, they propose to define a model of computation by means of an fUML model and

Extension	**Extension subject**	**Extension method**	**Extension purpose**
Laurent *et al.* [88]	Control and observation of executions	Extension of fUML execution model	Debugging
Tatibouet *et al.* [157]	Control of activity node execution order	fUML model library	Analysis through simulation
Benyahia *et al.* [9]	Control of activity node execution order	Extension of fUML execution model	Analysis of real time embedded systems through simulation
Model Driven Solutions [106]	Execution of SysML models: Streaming, time, structured models	Extension of fUML execution model	Analysis through simulation in systems engineering

Table 4.2: Overview of work related to fUML execution environment extensions

to delegate the execution control from an fUML model created for simulation purposes to this model. In the same way, non-functional aspects can be defined using fUML models. Therewith, a simulation library defining models of computation and non-functional aspects can be built. Thus, unlike our approach, the fUML virtual machine is not extended. However, the fUML model defining a system for simulation purposes has to be extended with calls to the simulation library, which is not required when using our command interface for execution control.

Benyahia *et al.* [9] also propose an extension of fUML supporting the control of the execution order of activity nodes. Similar to Tatibouet *et al.*, the aim of the authors is to enable the analysis of real time embedded systems through simulation based on fUML. For this purpose, they extend the fUML execution model with an explicit scheduler that controls the execution order of activity nodes. For defining the execution order, an additional semantic strategy class SelectNextActionStrategy is introduced into fUML's execution environment. As pointed out by Benyahia *et al.*, another important extension of the fUML execution model for enabling the analysis of real time embedded systems concerns time, as fUML does not define the semantics of time but allows the introduction of arbitrary time models. However, the authors leave this extension to future work.

The work of Tatibouet *et al.* [157] and the work of Benyahia *et al.* [9] should be understood as first steps towards providing the facilities for analyzing fUML models. They show that their extensions of fUML are technically feasible, but at the same time point out that more evidence is needed to also show their practical applicability.

In the course of the *Executable UML/SysML Semantics* project [106] carried out by Model Driven Solutions on behalf of the Lockheed Martin Corporation not only the reference implementation of the fUML virtual machine [91] has been developed, but also extensions of fUML for supporting systems engineering, in particular the execution of SysML models [116], have been elaborated. Particularly, the behavioral semantics of streaming inputs and outputs, time, and structured models has been considered and appropriate extensions of the fUML execution model have been proposed. Thereby, the behavioral semantics of time is related to our command interface incorporated into fUML's execution environment. For defining the behavioral semantics of time, it is proposed to extend the fUML execution model with an explicit threading model. A thread is a flow of execution, which may be suspended for a defined time duration and resumed after that duration. However, as reported by Model Driven Solutions [106], the implicit threading model of the fUML execution model as defined in the fUML standard [114] hinders the implementation of this explicit threading model. In particular, it is not possible to suspend an activity execution in case all threads of this activity execution are suspended and to resume it after a given time duration, because due to the implicit threading model, the activity execution terminates when all of its threads are suspended. This is because there is no explicit model of all threads of an activity execution defined in the fUML execution model. However, we believe, that our command interface would provide a solution for this problem as it maintains the state of ongoing activity executions and, thus, enables the suspension and resumption of activity executions.

CHAPTER 5

Semantics Specification with fUML

5.1 Design Rationale

The two key components of a modeling language's definition are the definition of the language's abstract syntax and the definition of the language's semantics (cf. Section 2.2). Formal definitions of these two components are required, in order to enable the automated processing of conforming models by means of computers.

For formally defining a modeling language's abstract syntax, metamodels are the standard means. MOF [121] constitutes a standardized, well established, and widely accepted metamodeling language enabling the definition of metamodels. The standardization of MOF fostered the emergence of techniques for efficiently developing tools operating on the syntax of models, such as techniques for automatically deriving modeling editors from metamodels, and generic components, such as components for model serialization, comparison, and transformation.

Unfortunately, no comparable standard means exist for formally defining a modeling language's behavioral semantics. This hampers the emergence of potential techniques for efficiently developing tools operating on the semantics of models, such as tools for model debugging, testing, and dynamic analysis [20]. However, having a standardized, well established, and widely accepted language for formally defining the semantics of modeling languages may—similarly to MOF—foster the emergence of such techniques.

We propose fUML to become this standardized, well established, and widely accepted semantics specification language. The following characteristics of fUML are in favor for the usage of fUML as semantics specification language in MDE. First of all, fUML, UML, and MOF are all standardized by OMG, and both fUML and MOF are based on UML. In particular, fUML and MOF both comprise UML class modeling concepts for defining structural aspects of systems and modeling languages, respectively. Thus, we argue that UML's activity modeling concepts and action language contained by fUML are likely to be suitable not only for defining behavioral aspects of systems, but also behavioral aspects of modeling languages. Furthermore, as fUML constitutes a subset of UML, and UML is the most adopted modeling language for MDE [70], the modeling concepts contained by fUML can be considered as being well known in the MDE

community. Another point in favor of fUML is its formally defined and standardized semantics in terms of a virtual machine that enables the execution of fUML conform models. Thus, when using fUML as a semantics specification language, models may be executed using the fUML virtual machine by executing the fUML activities defining the semantics of the used modeling language. Moreover, fUML is an imperative and object-oriented programming language, which constitute the predominant programming paradigms these days.

To enable the usage of fUML as semantics specification language for MDE, fUML has to be integrated with state of the art metamodeling languages, metamodeling methodologies, and metamodeling environments. Therefore, a *language integration strategy* for integrating fUML with existing metamodeling languages as well as a *semantics specification methodology* for systematically and efficiently developing executable semantics specifications with fUML are required. Furthermore, a *model execution environment* that enables the execution of models according to the used modeling language's fUML-based semantics specification and facilitates an efficient *development of semantics-based tools* has to be provided.

Integration with metamodeling languages. Existing metamodeling languages, such as MOF and Ecore, enable the formal definition of the abstract syntax of modeling languages, but do not provide any metamodeling concepts that enable the formal definition of the behavioral semantics of modeling languages. Furthermore, they do not provide an extension mechanism, such that additional metamodeling concepts could be introduced for this purpose. Thus, a strategy for integrating fUML with existing metamodeling languages is required enabling the definition of the behavioral semantics of modeling languages whose abstract syntax is defined in terms of metamodels conforming to the respective metamodeling language. This strategy has to take the modeling infrastructure provided by metamodeling languages into account, that is, in particular, the three-layered language definition hierarchy as introduced in Section 2.2.1.

Integration with metamodeling methodologies and environments. To foster a systematic and efficient development of behavioral semantics specifications, a methodology for developing behavioral semantics specifications with fUML is needed. This methodology should integrate seamlessly with existing methodologies and environments for developing modeling languages and only extend them concerning the specification of the behavioral semantics of modeling languages. Thus, the use of existing metamodeling techniques and tools, as well as any other techniques and tools for deriving modeling facilities or for processing models based on metamodels, shall not be affected. Further, the methodology shall enable a clear separation of the abstract syntax definition of a modeling language from its behavioral semantics definition and, thereby, enable a clear distinction between modeling concepts of the modeling language and runtime concepts that are solely needed for expressing the modeling language's behavioral semantics. Finally, the methodology shall enable the definition of the behavioral semantics of a modeling language with fUML in a way that enables the execution of conforming models by means of fUML's execution environment.

Support for model execution. To utilize the executability introduced into a modeling language by the definition of its behavioral semantics, a model execution environment is required.

In particular, based on the language definition of a modeling language, comprising the abstract syntax definition specified with an existing metamodeling language and the behavioral semantics definition specified with fUML, it shall be possible to execute conforming models. Therefore, the model execution environment shall leverage the standardized execution environment of fUML.

Facilitation of semantics-based tool development. As discussed beforehand, the model execution environment shall provide the means for executing models according to an fUML-based semantics specification of the respectively used modeling language. Furthermore, it shall provide means that facilitate the development of semantics-based tools building upon the executability of modeling languages. In particular, the implementation of model analysis methods and techniques for executable modeling languages, such as methods and techniques for model testing, dynamic analysis, debugging, and non-functional property analysis, shall be facilitated. Therefore, the extensions integrated with the fUML execution environment as discussed in Chapter 4 shall be made accessible in a way that enables the utilization of the provided event mechanism, command interface, and trace model in order to implement semantics-based tools. It has to be shown how these extensions of the fUML execution environment can be applied for developing semantics-based tools based on the elaborated model execution environment.

We investigated several strategies for integrating fUML with existing metamodeling languages resulting in the proposal of applying an integration-based operational approach leading to an executable metamodeling language that enables the definition of the abstract syntax and the behavioral semantics of modeling languages [98]. For this language, we elaborated a methodology comprising a set of processes, techniques and supporting tools that facilitates the development of semantics specifications and integrate seamlessly with existing metamodeling methodologies and metamodeling environments [99, 100]. We applied the proposed language integration strategy for integrating fUML with EMF's metamodeling language Ecore and implemented tool prototypes for EMF supporting the development of semantics specifications according to the elaborated methodology. Furthermore, we developed a model execution environment on top of the extended fUML execution environment that on the one hand supports the execution of models according to the used modeling language's fUML-based semantics specification and on the other hand provides the basis for efficiently developing semantics-based tools. For showing the latter, we implemented a model execution tool as well as a model debugger integrated with EMF on top of this model execution environment.

5.2 Semantics Specification Language

To facilitate the usage of fUML as a semantics specification language and, thereby, enable the definition of a modeling language's behavioral semantics using the modeling concepts provided by fUML, fUML has to be integrated with existing metamodeling languages. Therefore, distinct integration strategies can be applied, which we will discuss in Section 5.2.1. We assessed these distinct integration strategies concerning their suitability for defining the behavioral semantics of modeling languages as well as the possibility to integrate them with existing metamodeling

methodologies and metamodeling environments. Based on this assessment, we propose the application of the integration-based operational approach, which integrates fUML directly with an existing metamodeling language resulting in an executable metamodeling language. This language enables the specification of the behavioral semantics of modeling languages in an operational way as will be discussed in detail in Section 5.2.2.

5.2.1 Alternative Strategies for Integrating fUML with Metamodeling Languages

Translational vs. Operational Approach

To use fUML for formally defining the behavioral semantics of modeling languages, we may apply the *translational semantics approach* or the *operational semantics approach* as introduced in Section 2.2.2. In the following, we discuss in detail how these approaches can be applied to fUML as well as the advantages and disadvantages of these approaches.

Translational approach. In the translational approach, the behavioral semantics of a modeling language is defined through a translation of the modeling concepts provided by the considered modeling language to the modeling concepts provided by fUML. This translation between modeling languages and fUML can be implemented using model transformation languages. The fUML models resulting from the translation may be executed using the fUML virtual machine. However, when developing modeling languages having a semantics diverging from fUML's semantics, this approach has the disadvantage of potentially complex translations, which are difficult to specify due to the fact that three languages are involved, namely the modeling language, the transformation language, and fUML. An additional challenge arises with this approach when the results of executing the fUML models have to be traced back to the original models, which requires the translation of the execution results from fUML to the originally used modeling language.

In the fUML standard, the use of the translational approach for defining the behavioral semantics of the UML modeling concepts not contained by fUML is promoted [114, Subclause 7.1, p. 19]. In particular, it is suggested to use fUML as a *translational intermediary* between a surface subset of UML and target implementation languages.

Operational approach. Because of the drawbacks of the translational approach—the additional level of indirection introduced by the translation to fUML and the potentially high complexity of this translation—we advocate the operational approach. In this approach, fUML is used to define an interpreter for models conforming to the considered modeling language. This can be done either by directly introducing the definition of the interpreter into the modeling language's metamodel or by defining an interpreter separately from the modeling language's metamodel. The former case is achieved by adding operations to the metaclasses of the modeling language's metamodel and defining their behavior using fUML activities that specify the behavioral semantics of the respective modeling concept. In the latter case, a separate fUML model is created defining the structure and behavior of an interpreter, which takes as input a model to be executed and defines how this model is interpreted. In both cases, fUML is used as

action language for defining an interpreter of a modeling language. Having the behavioral semantics of a modeling language specified in terms of fUML activities, the fUML virtual machine can be used to execute models conforming to the considered modeling language.

The operational approach was applied for defining the behavioral semantics of fUML (cf. Section 3.3). In particular, in the fUML standard an execution model is specified using bUML—a subset of fUML—which defines an interpreter for fUML models. The execution model contains for each modeling concept of fUML a semantic visitor class that defines the behavioral semantics of the respective modeling concept. Thereby, the semantic visitor class of a modeling concept contains properties and associations for defining runtime concepts, as well as operations and activities defining how an instance of the respective modeling concept is executed. As pointed out before, the fUML standard suggests the application of the translational approach for defining the behavioral semantics of modeling concepts not contained by fUML. However, in the PSCS standard [122], the semantics of UML composite structures is defined by extending the fUML subset as well as the fUML execution model. This means, that the operational approach has been applied for defining the behavioral semantics of UML composite structures.

Transformation-based vs. Integration-based Operational Approach

As stated above, we advocate the operational approach for defining the behavioral semantics of modeling languages with fUML. For realizing this approach, we identified two distinct strategies that can be applied, namely a *transformation-based approach* and an *integration-based approach*. For discussing these alternative approaches in detail, we illustrate their application on the example of MOF in the following. However, please note that the following discussion applies not only to MOF but any metamodeling language, as they all provide metamodeling concepts similar to MOF's metamodeling concepts.

Before fUML can be used to define the behavioral semantics of modeling languages, whose abstract syntax is defined by MOF compliant metamodels, the gap between both modeling standards has to be bridged. Therefore, let us recap how fUML is composed and how it relates to MOF. For modeling structural aspects of a system, fUML contains a subset of the UML package Classes::Kernel. For modeling behavioral aspects of a system, fUML contains a subset of the UML packages CommonBehaviors, Activities, and Actions. As fUML uses the UML package Classes::Kernel and the same package is also merged into MOF, fUML's metamodel overlaps with MOF. Although fUML and MOF use the same elements for structural modeling on a conceptual level, they are on a technical level two distinct languages. This is shown in Figure 5.1 by depicting the language definition's of fUML and any MOF-based modeling language in terms of the metamodeling stack. While fUML models (*afUML Model*) are situated on the layer M1, the metamodel of a modeling language (*aML MM*) is located on the layer M2. However, an operational semantics specification of a modeling language—in our case the specification of an fUML model—has to reside on the same layer as the metamodel of the modeling language itself, as it has to define the behavioral semantics of the modeling concepts defined in the metamodel. To overcome the gap between MOF and fUML enabling the usage of fUML for specifying the behavioral semantics of modeling languages in an operational way, we identified the following two strategies.

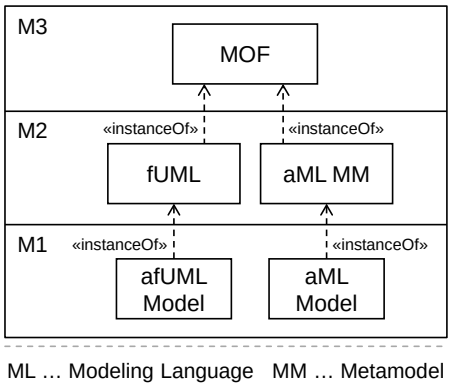


Figure 5.1: Language definitions of fUML and MOF-based modeling languages

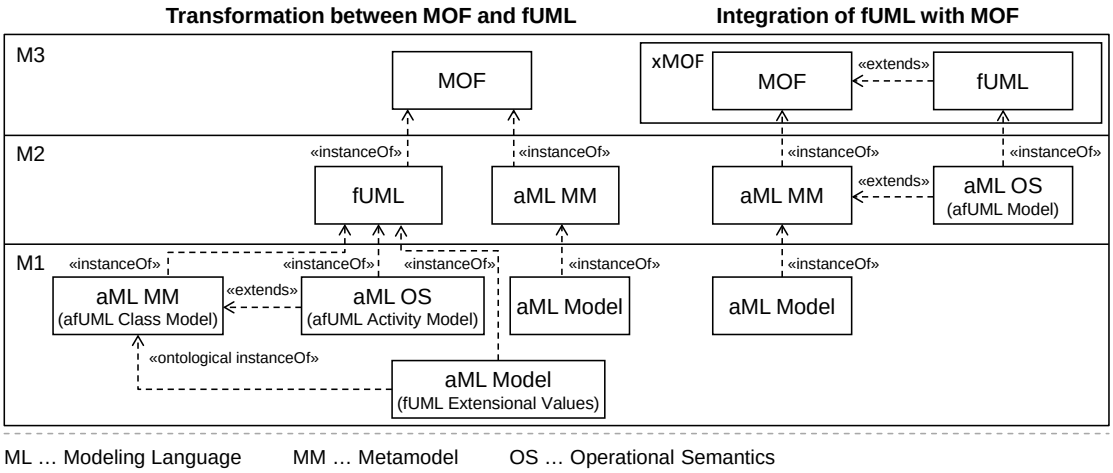


Figure 5.2: Strategies for using fUML as operational semantics specification language

Transformation-based approach. The first strategy is a transformation-based approach, depicted on the left-hand side of Figure 5.2. In this approach, the MOF-based metamodel of the modeling language (*aML MM* on M2) is transformed into an fUML model (*aML MM (afUML Class Model)* on M1) by mapping the elements of the MOF-based metamodel to elements of the fUML model, such that for each MOF metaclass a corresponding fUML class is created. As fUML contains the UML package Classes::Kernel to enable the definition of structural aspects of systems and this UML package is also merged into MOF for enabling the definition of metamodels, this transformation works straightforward. However, please note that the fUML standard excludes a few features of the package Classes::Kernel, which has to be taken into account by the transformation (cf. Subclause 7.2.2.1 and the paragraph *"Conventions on Derivation and Redefinition"* in Subclause 8.1 of the fUML standard [114]). The fUML classes resulting from the transformation can be extended for defining the behavioral semantics of the respective modeling

concept in terms of fUML activities (*aML OS (afUML Activity Model)* on M1). For executing a model conforming to the modeling language (*aML Model* on M1), it has to be transformed into fUML extensional values (*aML Model (fUML Extensional Values)* on M1) consisting of objects representing ontological instances [81] of the fUML classes that correspond to the modeling language's metaclasses and have been obtained by the transformation of the modeling language's metamodel. In case no metamodel is available in the first place, one may start on the layer M1 by purely using fUML for defining the abstract syntax and behavioral semantics of a modeling language and then generate a metamodel of the modeling language on the layer M2 afterwards.

One advantage of the transformation-based approach is that model transformations for metamodels and models are all that is needed to enable the operational definition of a modeling language's behavioral semantics and to enable the execution of models according to this definition. However, one major drawback is that both metamodeling environments and UML environments have to be used in parallel. A metamodeling environment has to be employed for defining the abstract syntax of a modeling language as well as conforming models, and a UML environment has to be used for defining the behavioral semantics of a modeling language as well as for executing models. Consequently, users have to switch between different environments and constantly apply transformations to metamodels and models to obtain equivalent fUML class diagrams and fUML object diagrams, respectively.

Integration-based approach. Because of the aforementioned drawbacks of the transformation-based approach, we advocate a second *integration-based* approach depicted on the right-hand side of Figure 5.2. In this approach, fUML is integrated with MOF by extending MOF with the behavioral part of fUML, i.e., its modeling concepts for defining activities as well as its action language, to obtain a new and executable metamodeling on layer M3 that we call *eXecutable MOF (xMOF)*. By extending MOF with certain parts of fUML, these parts are pulled up from the layer M2 to the layer M3 enabling the definition of the abstract syntax as well as of the behavioral semantics of a modeling language with one language that is composed of two standardized languages MOF and fUML. As a result, the abstract syntax can be specified in terms of a metamodel (*aML MM* on M2) using the modeling concepts provided by MOF, and the behavioral semantics (*aML OS* on M2) can be specified using fUML activities. A model conforming to the modeling language (*aML Model* on M1) can then be executed by employing the fUML virtual machine.

5.2.2 Executable Metamodeling Language xMOF

As discussed in Section 5.2.1, the operational approach for using fUML as semantics specification language has the following advantages over the translational approach. It requires no translation between the modeling concepts of a modeling language and the modeling concepts of fUML for specifying the behavioral semantics of the modeling language. Instead, using fUML an interpreter for the modeling language is defined, which defines how a conforming model is executed. Thus, no additional level of indirection is introduced in the semantics specification process and complex translation of metamodels and models to fUML as well as of execution results back to the modeling language can be avoided. For realizing the operational approach,

two distinct strategies can be applied. Thereby, the integration-based approach has the advantage over the transformation-based approach, that the definition of the modeling language's abstract syntax and behavioral semantics, the creation of models, as well as the execution of models can be performed in the used metamodeling environment instead of requiring to switch between metamodeling environment and UML environment. In the following, we show how the integration-based operational approach can be realized for integrating fUML with metamodeling languages. For doing so, we demonstrate the application of this integration strategy on the metamodeling language Ecore.

The integration-based operational approach for using fUML as a behavioral semantics specification language requires the integration of fUML's modeling concepts for defining behavioral aspects with the metamodeling language used for defining the abstract syntax of modeling languages. In particular, UML's activity modeling concepts as well as UML's action language contained by fUML have to be integrated with the metamodeling language. These modeling concepts have to be directly integrated with the meta-metamodel of the metamodeling language. Thereby, we aim at extending the meta-metamodel without having to actually modify it in order to not compromise any facilities relying on the meta-metamodel. For achieving this, we exploit the *reflexive definition* of metamodeling languages, meaning that the meta-metamodel of a metamodeling language is defined with the very same meta-metamodel and is, thus, an instance of itself. If we consider Ecore, its metamodel is defined using Ecore itself. For example, the metamodeling concept EClass is itself an instance of the metamodeling concept EClass. The same is also true for MOF. The metamodeling concept Class is merged into MOF's meta-metamodel from the UML metamodel, where it is defined as an instance of Class. A second characteristic of metamodeling languages that we exploit for the integration of fUML is their support of *inheritance*. In Ecore this is defined as a self-reference of the meta-metamodel element EClass. Thus, an instance of EClass defined in a metamodel may inherit features from arbitrary many other instances of EClass. In MOF, inheritance relationships are defined by the metamodeling concept Generalization, which can be used to define that a metaclass inherits features from arbitrary many other metaclasses.

Using these characteristics of metamodeling languages—their reflexive definition and support of inheritance—it is possible to integrate additional metamodeling concepts into a metamodeling language, without having to modify its meta-metamodel. Therefore, a new metamodel is created conforming to the respective metamodeling language, which defines the additionally required metamodeling concepts in terms of metaclasses, i.e., instances of the metamodeling concept provided by the respective metamodeling language for defining modeling concepts, such as EClass of Ecore and Class of MOF. For these metaclasses, inheritance relations are defined to elements of the metamodeling language's meta-metamodel for extending the respective metamodeling concept with additional features, such as properties, references, and operations.

By applying this technique, a metamodeling language can be extended with additional metamodeling concepts, in our case with concepts provided by fUML. The integration of fUML with a metamodeling language leads to a new language on the layer M3 being an executable metamodeling language, meaning that it enables the definition of the abstract syntax as well as the behavioral semantics of modeling languages. We refer to this resulting integrated language as *eXecutable MOF (xMOF)*.

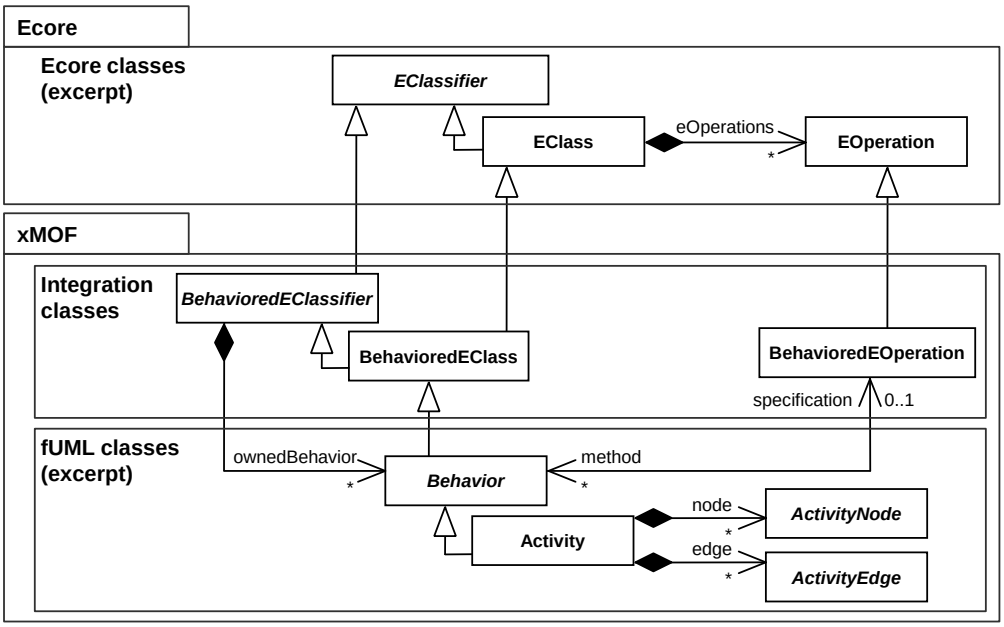


Figure 5.3: Excerpt of xMOF metamodel integrating fUML with Ecore

Figure 5.3 depicts how the integration of fUML with Ecore is realized using the discussed integration strategy. Please note that a corresponding extension can be applied also to MOF itself, instead of Ecore. For the integration of fUML with Ecore, we defined our own Ecore-based metamodel depicted as package labeled *"xMOF"*. In this metamodel, we defined the metaclasses BehavioredEClassifier, BehavioredEClass, and BehavioredEOperation labeled *"Integration classes"* that enable the integration of fUML with Ecore. These metaclasses are instances of the metamodeling concept EClass but have at the same time inheritance relationships to classes contained by Ecore's meta-metamodel, namely EClassifier, EClass, and EOperation, respectively. Therewith, they extend Ecore with additional metamodeling concepts. The metaclasses BehavioredEClassifier, BehavioredEClass, and BehavioredEOperation are defined according to the definition of the metaclasses BehavioredClassifier, Class, and Operation in fUML's metamodel. BehavioredEClassifier inherits from EClassifier and owns a containment reference to the metaclass Behavior being the supertype of the metaclass Activity. Thereby, the metaclasses Behavior and Activity are defined according to fUML's metamodel. Also all modeling concepts of fUML, which are required for defining activities, are added to the metamodel of xMOF and defined according to the metamodel of fUML. In particular, this applies to metaclasses defining different kinds of activity nodes and activity edges, but also to metaclasses defining value specifications as they are required by certain action types. Figure 5.3 depicts only an excerpt of the fUML metaclasses (labeled *"fUML classes"*) that are added to the metamodel of xMOF and therewith integrated into Ecore. Please note that any association defined in fUML's metamodel between the integrated metaclasses and the metaclasses contained by the fUML package Classes::Kernel have to be adapted, such that they refer to corresponding meta-metaclasses of Ecore or the integration classes BehavioredEClassifier, BehavioredEClass, and Behaviored-

EOperation defined by the metamodel of xMOF. For instance, the association between the fUML metaclasses CreateObjectAction and Classifier used for defining the classifier to be instantiated by the action is turned into a reference between the metaclass CreateObjectAction added to the metamodel of xMOF and the Ecore meta-metaclass EClassifier. Similarly, the association between the fUML metaclasses Behavior and BehavioredClassifier used for defining the context of a behavior is turned into a reference between the metaclass Behavior added to the metamodel of xMOF and the integration class BehavioredEClassifier defined in the metamodel of xMOF. Besides the metaclass BehavioredEClassifier, we defined the metaclass BehavioredEClass in xMOF's metamodel as a concrete subtype of EClass and BehavioredEClassifier. Thus, BehavioredEClass inherits all attributes and references of EClass and BehavioredEClass, in particular the containment references to EOperation and Behavior. The metaclass BehavioredEOperation enables the specification of the behavioral semantics of metaclasses by defining the behavior of metaclass operations. Therefore, an inheritance relationship is defined to Ecore's metaclass EOperation as well as a reference to the metaclass Behavior.

With xMOF it is now possible to use fUML activities (metaclass Activity) to specify the behavior of operations (metaclass BehavioredEOperation) owned by metaclasses (metaclass BehavioredEClass) in the metamodel of a modeling language. Hence, it is possible to specify the behavioral semantics of modeling languages in an operational way.

5.3 Semantics Specification Methodology

With xMOF, as introduced in the previous section, the abstract syntax as well as the behavioral semantics of modeling languages can be defined. To foster a systematic and efficient development of behavioral semantics specifications with xMOF, as well as to maximize the reuse and the compatibility with existing metamodeling methodologies and metamodeling environments, we propose a dedicated methodology, which is orthogonal to existing metamodeling methodologies (e.g., [75, 77, 149, 150, 156]).

The proposed methodology consists of the executable metamodeling language xMOF and a set of processes, techniques, and supporting tools for specifying the behavioral semantics of modeling languages as well as for executing models based on the behavioral semantics specifications. The processes of the methodology are depicted in Figure 5.4 showing which artifacts are produced and consumed by the individual process steps, as well as information about which process steps are automated and which are manually performed. The methodology consists of the following four processes.

- **Language design.** In the *language design* process, the modeling language is developed by defining the language's abstract syntax by means of a metamodel conforming to an existing metamodeling language, such as Ecore, as well as the language's behavioral semantics by means of xMOF. Thereby, the metamodel builds the basis for specifying the behavioral semantics of a modeling language, which is defined in a separate artifact referred to as an *xMOF-based configuration*.

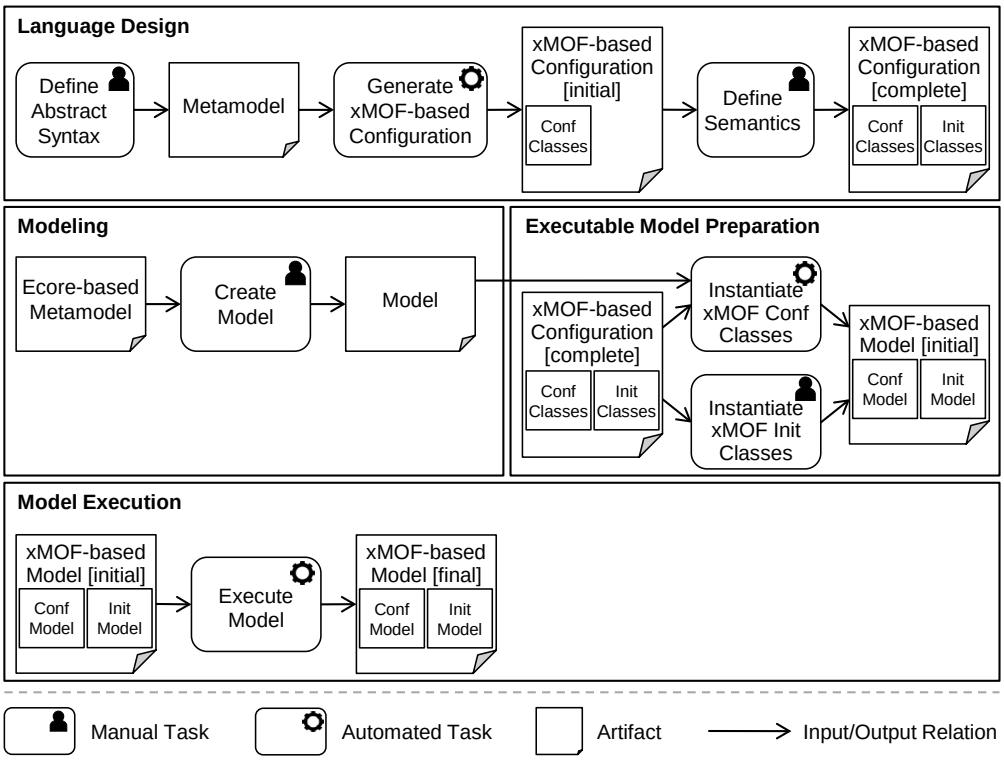


Figure 5.4: xMOF semantics specification methodology

- **Modeling.** In the *modeling* process, a model of the beforehand developed modeling language is created. Therefore, modeling facilities available for the respective modeling language may be used.

- **Executable model preparation.** As with xMOF the behavioral semantics of the modeling language is formally defined with fUML activities, models conforming to the modeling language may be executed by utilizing the fUML virtual machine. Therefore, the model to be executed must be first transformed into an executable form referred to as an *xMOF-based model*, that is an instance of the xMOF-based configuration defining the behavioral semantics. Furthermore, additional input may be required for executing the model, which is specified manually in the xMOF-based model. This is done in the *executable model preparation* process.

- **Model execution.** In the *model execution* process, the model is finally executed according to the xMOF-based semantics specification. From the model execution, the final runtime state of the executed model, that is the state of the xMOF-based model after its execution finished, can be obtained and provided as feedback to the modeler as well as further processed, for instance, for analysis purposes.

The four processes of our methodology for defining executable modeling languages based on

fUML are in the following described in more detail. Therefore, we make use of the Petri net language as running example and show how its behavioral semantics can be formally defined using xMOF as well as how conforming models can be executed according to this behavioral semantics specification.

Language design. In the language design process, the language designer first defines the *abstract syntax* of the modeling language by means of a *metamodel*. Therefore, an existing metamodeling language, such as Ecore, is used.

The behavioral semantics of the modeling language is then defined using xMOF in the so-called *xMOF-based configuration*. Therefore, an xMOF-based configuration is automatically initialized from the metamodel. In particular, for each concrete metaclass in the metamodel, one BehavioredEClass instance is generated referred to as a *configuration class*. Each of these generated configuration classes is defined as subclass of the respective metaclass and is extended by the language designer to define the behavioral semantics of this metaclass. Therefore, additional attributes, references, and configuration classes may be added for defining runtime concepts required for capturing the state of an executing model. For expressing the actual behavior of the modeling language's concepts, operations (xMOF metaclass BehavioredEOperation) and their implementations in terms of activities (xMOF metaclass Activity) are added to the configuration classes. Thereby, one main operation has to be defined, which—like, for instance, in Java programs—serves as entry point for executing a model according to the semantics specification.

Furthermore, supplementary data required as additional input for the execution of a model may be defined in the xMOF-based configuration. Therefore, additional instances of the xMOF class BehavioredEClass may be defined referred to as *initialization classes*.

The xMOF-based configuration consisting of configuration classes, initialization classes, and their respective behavior specifications completely define the behavioral semantics of a modeling language.

Example. The metamodel of the Petri net language is depicted in Figure 5.5. A Petri net (metaclass Net) consists of a set of uniquely named places (metaclass Place) and transitions (metaclass Transition), where transitions reference their input and output places (references input and output).

The xMOF-based configuration defining the behavioral semantics of the Petri net language consist of four configuration classes and one initialization class as shown in Figure 5.6. The configuration classes NetConfiguration, TransitionConfiguration, and PlaceConfiguration were automatically initialized for the metaclasses Net, Transition, and Place, respectively. The configuration class Token was additionally defined for representing tokens residing at places during the execution of Petri net models (references heldTokens and holdingPlace). The class TokenDistribution constitutes an initialization class allowing the definition of a set of tokens (reference initialTokens) serving as initial token distribution of a Petri net. This set of tokens constitutes the input required for executing a Petri net model.

For defining the behavioral semantics of the modeling concepts offered by the Petri net language, operations were introduced into the configuration classes NetConfiguration, TransitionConfiguration, and PlaceConfiguration. Their behavior is defined by the activities shown in Fig-

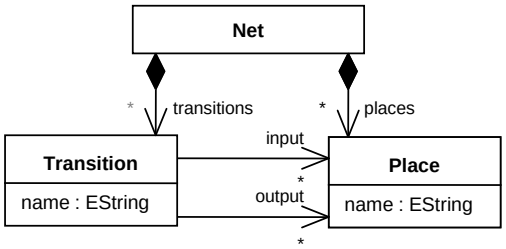


Figure 5.5: Semantics specification example: Ecore-based metamodel

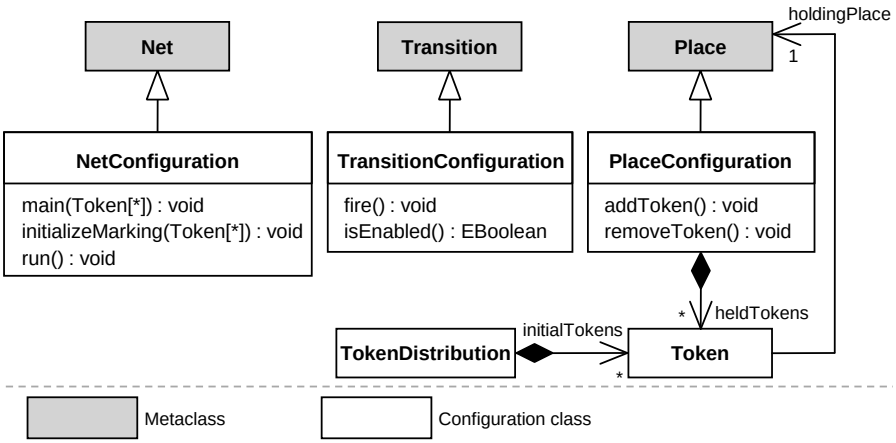


Figure 5.6: Semantics specification example: xMOF-based configuration (classes)

ure 5.7. The main() operation of NetConfiguration serves as entry point for executing a Petri net model. It first calls the operation initializeMarking(), which initializes the heldTokens reference of the Place instances according to the set of tokens provided as input, before the operation run() is invoked. The operation run() collects in a loop the set of enabled transitions by invoking the operation isEnabled() for each transition of the net. A transition is enabled when all input places of the transition hold at least one token. Therefore, the operation isEnabled() collects all input places of a transition holding no token and returns false if at least one such place holding no token was collected and true if no such place was collected. The operation run() then selects the first enabled transition and invokes the operation fire() for this transition. Subsequently, the operation fire() calls for each output place of the transition the operation addToken() to create a new token residing at the respective output place and removeToken() for each input place to destroy one token residing at the respective input place.

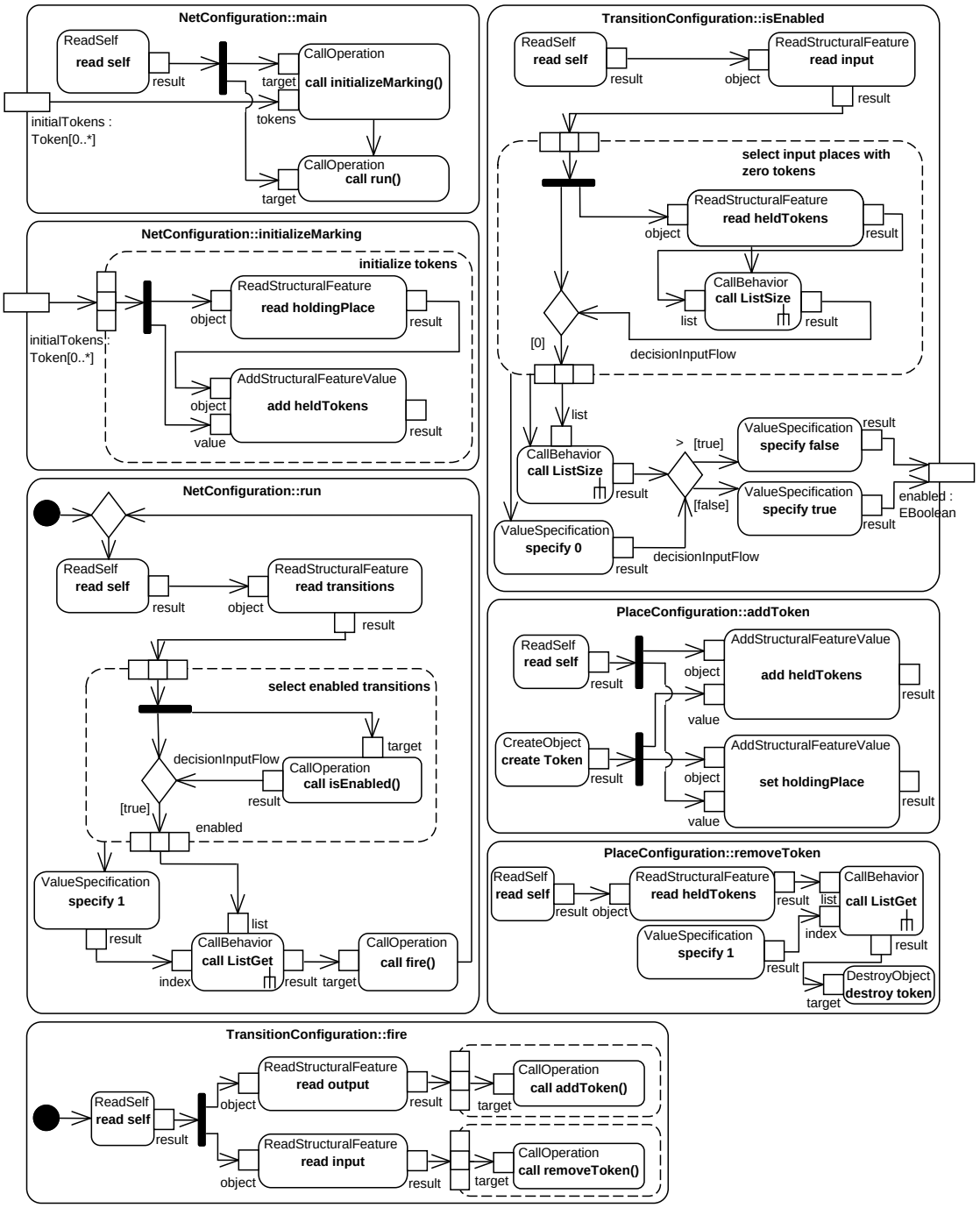


Figure 5.7: Semantics specification example: xMOF-based configuration (activities)

Modeling. Using existing modeling facilities, a user may create *models* conforming to the defined modeling language by instantiating the metamodel defined in the language design process. Thus, the creation of models is not affected by our methodology at all and any modeling editor may be used to conveniently create models in a graphical or textual notation. In the EMF metamodeling environment, such editors might be developed, for instance, with GMF[1], Graphiti[2], or Xtext[3].

Example. On the top left-hand side of Figure 5.8, an example of a Petri net model is depicted in graphical notation. It consists of two places *p1* and *p2* and one transition *t*. Thereby the place *p1* constitutes the input place of the transition *t* and the place *p2* its output place.

Executable model preparation. As with xMOF the behavioral semantics of a modeling language is defined by fUML activities, conforming models can be executed by leveraging the fUML virtual machine. Therefore, the model to be executed has to be first represented in an executable form, that is in terms of an instance of the xMOF-based configuration defining the behavioral semantics of the respective modeling language. This instance is referred to as an *xMOF-based model*. Therefore, for each element in the model, the configuration class defining the behavioral semantics of the respective modeling concept is instantiated and initialized automatically. The initialization includes the setting of attribute values as well as references of the configuration class instances according to the attribute values and references of the model elements. The obtained model is referred to as a *configuration model*. In addition, the initialization classes have to be instantiated by the modeler in an *initialization model* to define the additional input required for executing the model. Together with this initialization model, the configuration model is ready for being executed.

Example. On the bottom left-hand side of Figure 5.8, the xMOF-based model instantiated for the Petri net model created in the beforehand discussed modeling process is depicted. For the net *n*, the transition *t*, and the two places *p1* and *p2*, the respective configuration classes NetConfiguration, TransitionConfiguration, and PlaceConfiguration were instantiated and the resulting instances were initialized according to the model elements (cf. top left-hand side of Figure 5.8). As input to the model execution, the initialization class TokenDistribution was manually instantiated leading to the TokenDistribution instance *td* and one Token instance *t1* residing at place *p1* was added constituting the initial token distribution of the net.

Model execution. Having obtained an xMOF-based representation of the model to be executed, we may perform the execution by leveraging the fUML virtual machine. A detailed description of how the model execution is achieved is provided in Section 5.4. The result of the execution is the final runtime state of the *xMOF-based model*, which has been manipulated during the model execution according to the xMOF-based behavioral semantics specification

[1] `http://www.eclipse.org/modeling/gmp`, accessed 24.07.2014
[2] `https://www.eclipse.org/graphiti`, accessed 28.07.2014
[3] `http://www.eclipse.org/Xtext`, accessed 24.07.2014

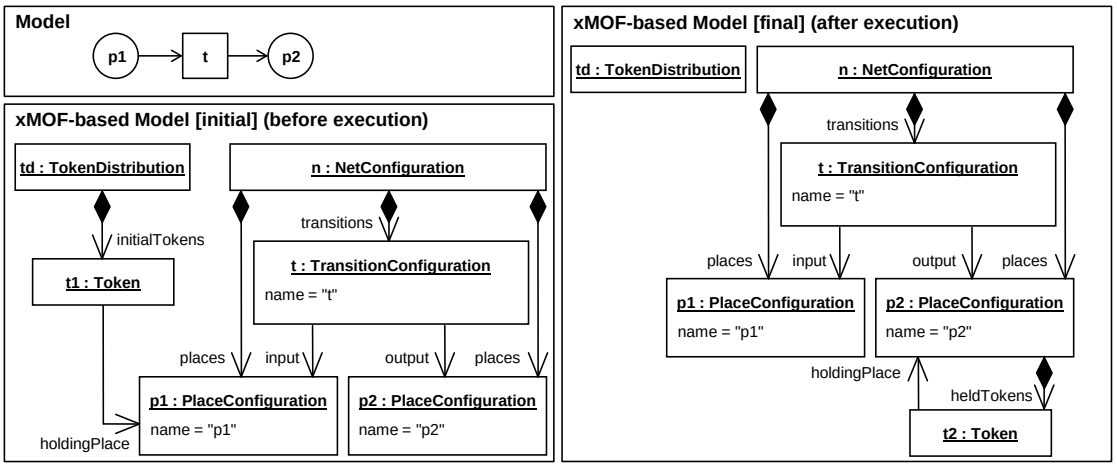


Figure 5.8: Semantics specification example: Model and xMOF-based model

defined for the used modeling language. This final xMOF-based model can be presented to the user as execution result and further processed for analysis purposes.

Example. The final xMOF-based model resulting from executing the example Petri net model is depicted on the right-hand side of Figure 5.8. During the execution, the token *t1*, which resided at place *p1* in the initial token distribution *td* (cf. bottom left-hand side of Figure 5.8) was destroyed as a result of the firing of transition *t*, and a new token *t2* was added to the place *p2*.

Tool Support. We provide an implementation of the presented methodology for EMF consisting of a set of Eclipse plugins that support language designers in defining the behavioral semantics of modeling languages with xMOF as well as modelers in executing models according to xMOF-based semantics specifications. For the language design process, we provide tool support for generating xMOF-based configurations from Ecore-based metamodels as well as editing support for extending generated xMOF-based configurations with configuration classes, initialization classes, and activities for specifying the behavioral semantics of modeling languages. For the executable model preparation process, we provide tool support for instantiating initialization classes as well as defining input required for executing models. The tool support provided for the model execution process will be discussed in Section 5.4. The mentioned tools are briefly discussed in the following.

- **xMOF-based configuration generation.** For creating xMOF-based configurations, we provide a wizard allowing to conveniently initialize an xMOF-based configuration for an Ecore-based metamodel. Therefore, the user has to provide the following input. The Ecore-based metamodel, the metaclass that should own the main operation of the be-

havioral semantics specification, as well as the file system location of the xMOF-based configuration to be created.

- **Semantics specification with xMOF.** The behavioral semantics of a modeling language is specified by extending the automatically initialized xMOF-based configuration with additional configuration classes, initialization classes, attributes, references, operations, and activities. Therefore, we provide an editor allowing the definition of classes in a tree-based notation and activities in a graphical notation adopting the standardized graphical activity diagram notation of UML. A screenshot of this editor is depicted in Figure 5.9. On the upper left-hand side, the tree-based editor for defining classes in xMOF-based configurations is shown. It depicts the configuration classes and initialization classes as well as their attributes, references, operations, and activities. The activities themselves can be edited in a graphical editor as shown at the bottom of Figure 5.9. Properties of elements can be modified as usual in the *properties view* depicted on the upper right-hand side.

- **Initialization class instantiation.** For instantiating initialization classes contained by an xMOF-based configurations we provide a wizard. In this wizard the user selects the xMOF-based configuration, the initialization class to be instantiated, as well as the file system location of the initialization model to be created. The created initialization model can then be extending using a tree-based editor to define the input for the execution of a model.

- **Input definition.** For defining the input parameter values to be provided to the main operation of the xMOF-based configuration when executing a model, we again provide a wizard. The user selects the xMOF-based configuration, the main operation, as well as the file system location of the parameter value definition file to be created. In this file, the user defines references to elements of the initialization model for defining the values to be provided to the model execution.

The proposed methodology may be used *orthogonally to existing methodologies* for developing modeling languages. The usual steps for developing the abstract syntax and concrete syntax of a modeling language, as well as for creating models are not affected. Consequently, the facilities provided by existing metamodeling environments that operate on the syntax of a modeling language, such as editors for creating metamodels, generators for modeling editors, model transformation and code generation engines, remain unaffected by the methodology and can be applied as usual. We extended the metamodeling environment EMF with appropriate *tool support* for the elaborated methodology enabling the application of its provided processes and techniques for specifying the behavioral semantics of modeling languages and executing models based on fUML in any project that makes use of Ecore-based metamodels.

With the proposed methodology, the definition of the abstract syntax and the definition of the behavioral semantics of a modeling language are *clearly separated* from each other in distinct artifacts. Thus, the metamodel does not incorporate any semantics-specific aspects and solely defines the abstract syntax of the modeling language. Semantics-specific aspects are defined in the xMOF-based configuration consisting of configuration classes, initialization classes, and

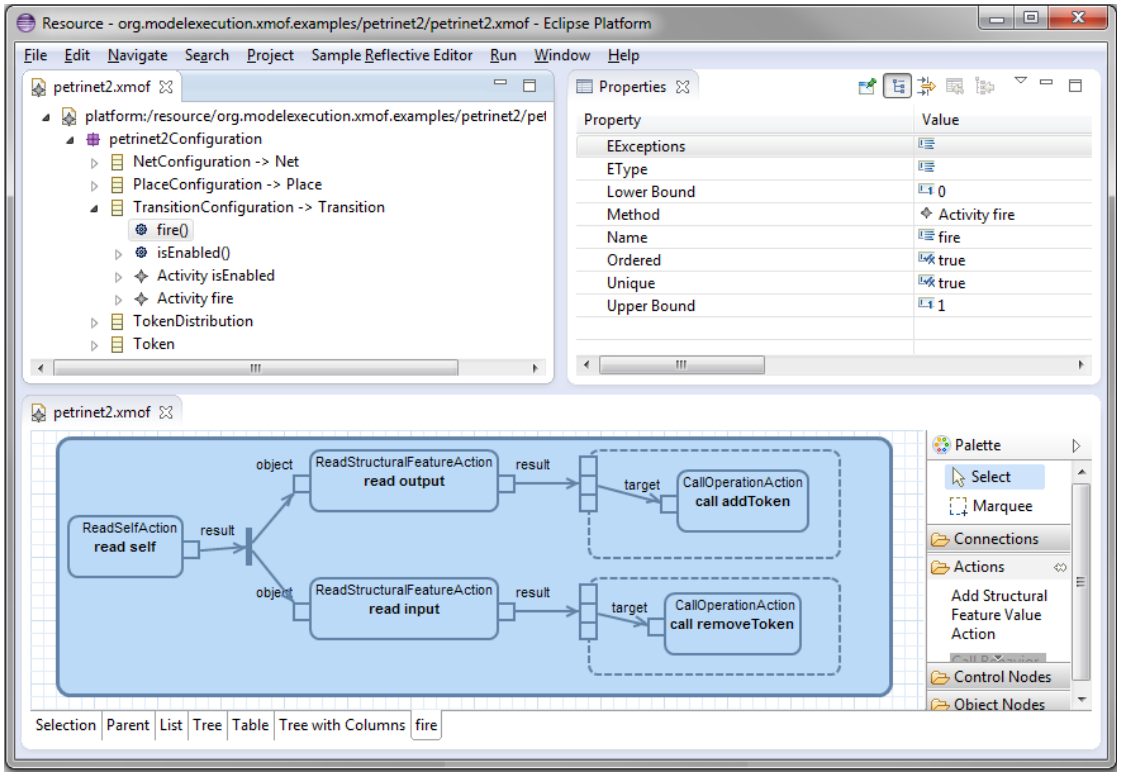


Figure 5.9: Screenshot of xMOF semantics specification editor

activities. This separation also allows the definition of *multiple behavioral semantics* for the same metamodel by defining multiple xMOF-based configurations. The xMOF-based configuration of a modeling language constitutes on the one hand the formal definition of the language's behavioral semantics, but on the other hand also acts as representation for the *runtime state* of executed models consisting of runtime variables, such as the tokens residing at places in the defined Petri net language. Moreover, the xMOF-based configuration allows the separate definition of *additional and potentially complex input parameter values* that are needed for executing models, such as the initial token distribution in the Petri net language. By leveraging the fUML virtual machine, *models can be directly executed*. Only the behavioral semantics, the model to be executed, and the input parameter values have to be provided for executing a model.

5.4 Model Execution Environment

When applying the semantics specification methodology for xMOF as discussed in the previous section, the abstract syntax of a modeling language is defined in terms of a metamodel conforming to an existing metamodeling language and the behavioral semantics of a modeling language is defined in terms of an xMOF-based configuration conforming to xMOF. Thereby, the xMOF-

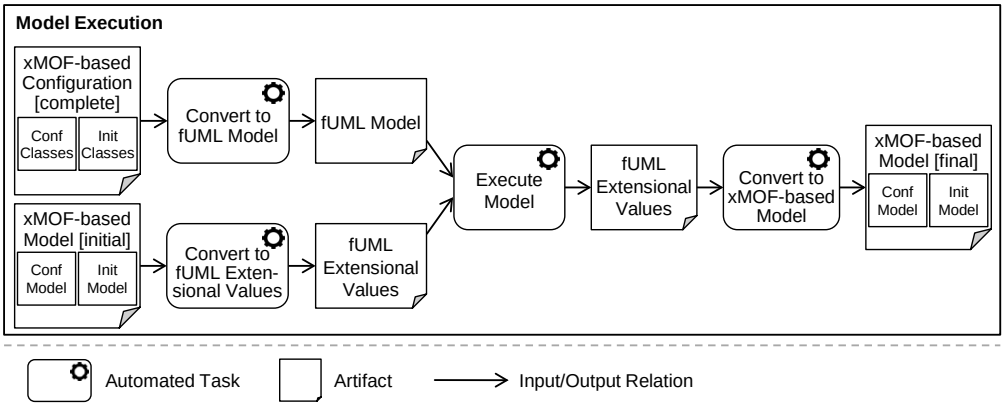


Figure 5.10: Model execution process performed by generic model execution environment

based configuration of a modeling language defines the behavioral semantics of the provided modeling concepts in terms of fUML conform activities. This enables the utilization of fUML's execution environment for executing models according to the used modeling language's behavioral semantics as defined in its xMOF-based configuration. Therefore, the model to be executed has to be first transformed into a configuration model, which is an instance of the xMOF-based configuration—in particular of its configuration classes—and constitutes the initial runtime state of the model. Furthermore, the user may create an initialization model by instantiating initialization classes defined in the xMOF-based configuration for defining additional input for the model to be executed. The xMOF-based model, comprising the configuration model and the initialization model, constitutes an executable representation of the model to be executed providing all information necessary for performing the model execution.

The model execution is performed by the generic *model execution environment*, which we developed based on the extended fUML execution environment presented in Chapter 4. Figure 5.10 details the *model execution* process of our semantics specification methodology for xMOF (cf. Figure 5.4). The shown process steps are carried out by the developed model execution environment. To utilize the extended fUML execution environment for executing models, the xMOF-based configuration defining the behavioral semantics to be applied for the model execution as well as the xMOF-based model constituting the executable representation of the model to be executed first have to be converted to fUML. The xMOF-based configuration has to be converted into an fUML model and the xMOF-based model has to be converted into fUML extensional values. For that purpose, we provide an *xMOF-to-fUML converter* performing this conversion automatically. The fUML model obtained from the conversion of the xMOF-based configuration is then *executed by means of the extended fUML execution environment* whereat the fUML extensional values obtained from the conversion of the xMOF-based model serve as input to the execution. During the execution, the fUML extensional values representing the executing model are manipulated by the fUML virtual machine according to the semantics specification represented by the executing fUML model. Thus, after the execution of the fUML model is completed, the fUML extensional values represent the final runtime state of the executed model. These fUML extensional values are converted back to xMOF by an *fUML-to-xMOF converter*

resulting in an xMOF-based model capturing the final runtime state of the executed model. The steps of the model execution process are in the following described in more detail.

xMOF to fUML conversion. In the conversion of the xMOF-based configuration to an *fUML model*, each configuration class and initialization class contained by the xMOF-based configuration is converted into a corresponding fUML class including attributes, references, and operations. Furthermore, the activities contained by the xMOF-based configuration defining the behavior of the classes' operations are converted into corresponding fUML activities. The conversion of activities from xMOF to fUML is a straightforward one-to-one transformation, as the language concepts provided by xMOF for defining activities are adopted from fUML without modifications and are, hence, identical to the metaclasses defined by fUML's metamodel. For defining configuration classes and initialization classes as well as their operations, xMOF extends the metamodeling concepts for defining metaclasses and operations provided by the respectively used metamodeling language in order to enable the definition of their behavior using fUML conform activities (cf. Section 5.2.2). In the case of Ecore, the metamodeling concepts EClass and EOperation are extended by xMOF through the introduced subtypes BehavioredEClass and BehavioredEOperation. Thus, configuration and initialization classes are instances of BehavioredEClass and their operations are instances of BehavioredEOperation. For defining attributes and references of configuration classes and initialization classes the metamodeling concepts provided by the used metamodeling language are retained by xMOF. Hence, in the case of Ecore, they are defined using the metamodeling concepts EAttribute and EReference. Due to the inheritance relationships defined between configuration classes contained by the xMOF-based configuration and metaclasses contained by the metamodel of the modeling language, also the metaclasses and their attributes, references, and operations being instances of EClass, EAttribute, EReference, and EOperation are converted to fUML. In the conversion of an xMOF-based configuration and the extended metamodel, instances of EClass and BehavioredEClass, EOperation and BehavioredEOperation, EAttribute, as well as EReference are converted into instances of fUML's metaclasses Class, Operation, Property, and Association, respectively. This conversion is straightforward to implement as there are no conceptual differences between metamodeling concepts and UML class modeling concepts.

In the conversion of an xMOF-based model to *fUML extensional values*, the model elements contained by the xMOF-based model are converted into fUML objects, the model elements' attribute values are converted into fUML feature values, and the references between the model elements are converted into fUML links. Thereby, the objects, feature values, and links are instances of the classes, properties, and associations contained by the fUML model obtained from the conversion of the xMOF-based configuration.

Model execution by fUML execution environment. Using the command interface of the extended fUML execution environment, the fUML extensional values obtained from the conversion of the xMOF-based model are first added to the locus of the fUML execution environment before the execution of the fUML model obtained from the conversion of the xMOF-based configuration is started. Thereby, the execution is started at the activity defined for the main operation of the xMOF-based configuration. The xMOF-based model element being an instance

of the configuration class owning the main operation is provided as context object for executing this activity, and the fUML extensional values created during the conversion of the initialization model are provided as input to this activity.

fUML to xMOF conversion. During the execution of the fUML model obtained from the conversion of the xMOF based configuration, the fUML virtual machine interprets the fUML activities defining the behavioral semantics of the modeling language and manipulates the fUML extensional values representing the model to be executed accordingly. The result of the execution consists in the runtime state of these fUML extensional values after the execution finished constituting the final runtime state of the executed model. The fUML extensional values may be converted back to an *xMOF-based model* by converting each fUML object to an instance of the respective configuration class and each fUML link between two fUML objects to references between these configuration class instances. Thus, the user may be provided with the final runtime state of the executed model in terms of an xMOF-based model and this runtime state may be further processed for analysis purposes.

5.5 Semantics-based Tool Development

The model execution environment presented in the previous section enables the execution of models conforming to any modeling language whose behavioral semantics is defined with xMOF. Therefore, as depicted in Figure 5.11, the definition of the modeling language comprising the definition of the language's abstract syntax in terms of a metamodel as well as the definition of the language's behavioral semantics in terms of an xMOF-based configuration is converted into an fUML model. Similarly, the model to be executed as well as additional input to the model both contained by an xMOF-based model are converted into fUML extensional values. The obtained fUML model is then executed for the obtained fUML extensional values by means of fUML's execution environment. Besides the final runtime state of the executed model, further runtime information can be obtained from the model execution. As depicted in Figure 5.11,

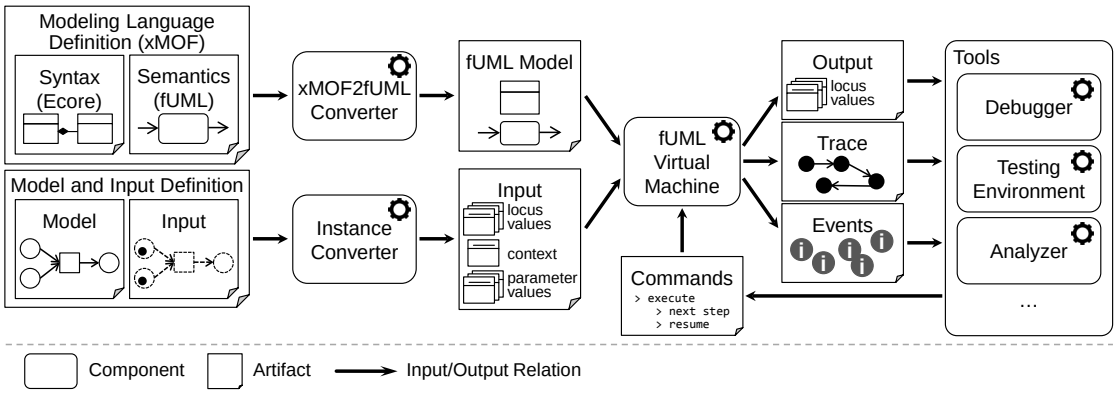


Figure 5.11: Model execution environment for xMOF

the event mechanism, command interface, and trace model introduced into the fUML execution environment can be utilized to observe, control, and analyze model executions. Based on these capabilities, *semantics-based tools*, such as debuggers, testing environments, and dynamic model analysis tools can be implemented on top of the developed generic model execution environment.

By utilizing the generic model execution environment for xMOF in this way, we developed a model execution tool as well as a model debugger. The *model execution tool* uses the model execution environment for executing models and annotates the executed model with the execution result, that is the final runtime state of the model obtained from the model execution. The *debugger* makes use of the command interface and event mechanism provided by the model execution environment to enable the stepwise execution of models. Thereby, after the completion of each execution step, the current runtime state of the debugged model is again annotated on the model. In the following, we present the implementations of these two developed semantics-based tools in further detail.

5.5.1 Model Execution

The model execution tool is implemented as an Eclipse plugin on top of the model execution environment for xMOF presented in Section 5.4. It provides tool support for executing models based on the used modeling language's xMOF-based semantics specification. Therefore, it provides a launch configuration enabling the user to start the execution of a model by selecting the model to be executed, the xMOF-based configuration to be used for the model execution, as well as additional input to the model execution. Furthermore, the model execution tool enables users to inspect the result of a performed model execution in a comprehensible way by annotating the executed model with the execution result. Therefore, it makes use of the Eclipse plugin EMF Profiles [87]. EMF Profiles constitutes an adaptation of the UML profile mechanism for EMF and, therewith, a lightweight extension mechanism for modeling languages defined with Ecore. In particular, it allows the definition of profiles consisting of stereotypes that can be applied to model elements for annotating additional information on models. Applied profiles can be loaded into the modeling editor of the used modeling language for visualizing the information provided by stereotype applications. Thereby, EMF Profiles supports loading profile applications into tree-based editors created with EMF as well as graphical editors implemented with GMF and Graphiti. In order to utilize EMF Profiles for annotating the result of a model execution on the executed model, the model execution tool generates from the used modeling language's definition a profile conforming to EMF Profiles. After a model execution is completed, an application of this profile is generated from the execution result and loaded on the executed model to provide the execution result to the user in a comprehensible way. These two steps referred to as *runtime profile generation* and *runtime profile application* are depicted in Figure 5.12 and in the following discussed in further detail.

Runtime profile generation. The output of a performed model execution consists in the final runtime state of the executed model, that is the final runtime state of the elements contained by the executed model. With xMOF, the runtime state of a model element is composed of the

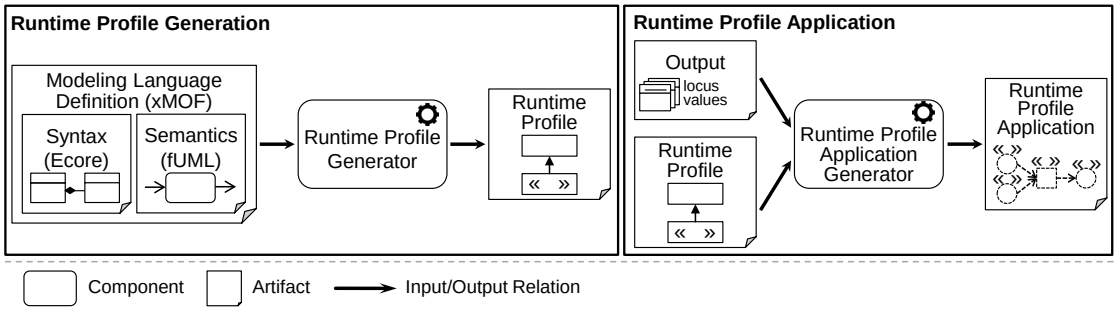


Figure 5.12: Model execution tool

element's attribute values as well as the element's references to other model elements. Only attributes and references defined by the configuration class of the respective modeling concept are of interest, as only their values are manipulated during the execution of a model, whereas values of attributes and references defined by the respective metaclass are constant. Thus, for annotating the runtime state of a model, we need a way to capture the model elements' values for attributes and references defined by configuration classes. For this purpose we generate from the configuration classes defined by a modeling language's xMOF-based configuration a so-called *runtime profile* conforming to EMF Profiles. In particular, for each configuration class we generate a dedicated stereotype comprising tagged values and references corresponding to the attributes and references of the respective configuration class. This enables us to annotate values for these attributes and references on a model and, thus, to annotate the final runtime state of the model. The generation of the runtime profile from an xMOF-based configuration is performed automatically. For starting the generation process, we provide a wizard integrated with Eclipse that requires the user only to select the xMOF-based configuration of the modeling language.

Example. The runtime profile generated for the xMOF-based configuration of the Petri net language is depicted in Figure 5.13. It consists of the stereotypes NetConfigurationStereotype, TransitionConfigurationStereotype, and PlaceConfigurationStereotype. These stereotypes are applicable to model elements being instances of the metaclasses Net, Transition, and Place, respectively. The attributes and references defined for the configuration classes are accordingly introduced into the stereotypes as tagged values and references, respectively. In particular, the containment reference heldTokens owned by the configuration class PlaceConfiguration and referring to the configuration class Token (cf. Figure 5.6) is introduced into the stereotype PlaceConfigurationStereotype. By applying the stereotype PlaceConfigurationStereotype to places contained by a Petri net model and adding Token instances to these stereotype applications, it is possible to annotate the token distribution of the Petri net directly on the respective Petri net model.

Runtime profile application. Being equipped with the runtime profile of an xMOF-based configuration, it is possible to annotate the final runtime state of an executed model directly on

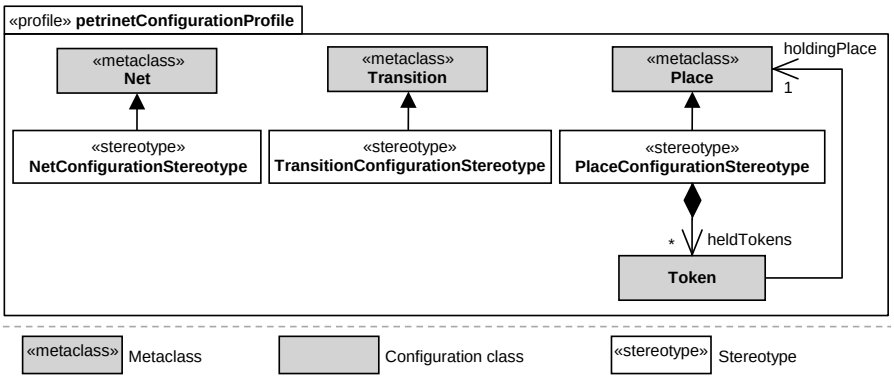


Figure 5.13: Model execution example: Runtime profile

this model. Therefore, an application of the runtime profile on the executed model is generated from the output of the model execution, that consists in the fUML extensional values residing at the locus of the execution performed by the fUML virtual machine, which represent the final runtime state of the model. This profile application is referred to as a *runtime profile application*. Thereby, on each model element the stereotype defined for the element's metaclass is applied and the resulting stereotype application's tagged values and references are set according to the feature values and links of the fUML object representing the final runtime state of the respective model element. Alternatively, the runtime profile application could be generated from the final xMOF-based model obtained from the model execution instead of directly from the fUML extensional values. The generated profile application may be loaded into the modeling editor for inspecting the final runtime state of the executed model.

Example. Figure 5.14 shows the runtime profile application generated for the example Petri net model (cf. Figure 5.8). The stereotype NetConfigurationStereotype was applied to the net *n*, the stereotype TransitionConfigurationStereotype was applied to the transition *t*, and the stereotype PlaceConfigurationStereotype was applied to the places *p1* and *p2*. Furthermore, the token *t2* was added to the stereotype application of the place *p2*. Thus, the generated runtime profile application captures the final runtime state of the executed model, that is the token distribution of the modeled Petri net consisting of one token residing at place *p2*.

Figure 5.15 depicts a screenshot of the implemented model execution tool showing how generated runtime profile applications are visualized by EMF Profiles. On the top left-hand side the GMF-based editor of the Petri net language can be seen. It shows the executed Petri net model in graphical concrete syntax. The place *p2* is highlighted by a green border and an overlayed icon depicting a token, which indicates that the stereotype PlaceConfigurationStereotype is applied to this place and that this stereotype application contains at least one Token instance. On the top right-hand side the *EMF Profile applications view* is depicted, which is responsible for displaying the stereotypes applied to the model opened in the currently active editor. It shows the stereotypes applied to currently selected model elements. As the place *p2* is selected in the Petri net editor, the *EMF Profile applications view* shows its application of the PlaceConfiguration-

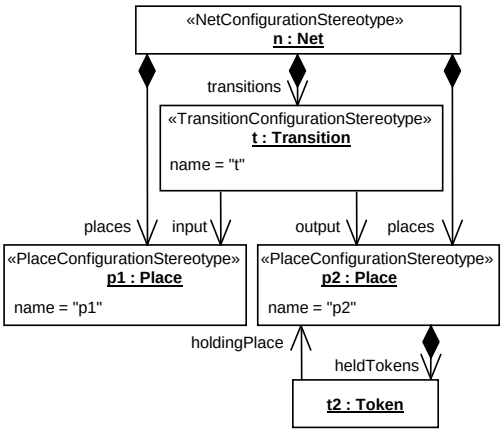


Figure 5.14: Model execution example: Runtime profile application

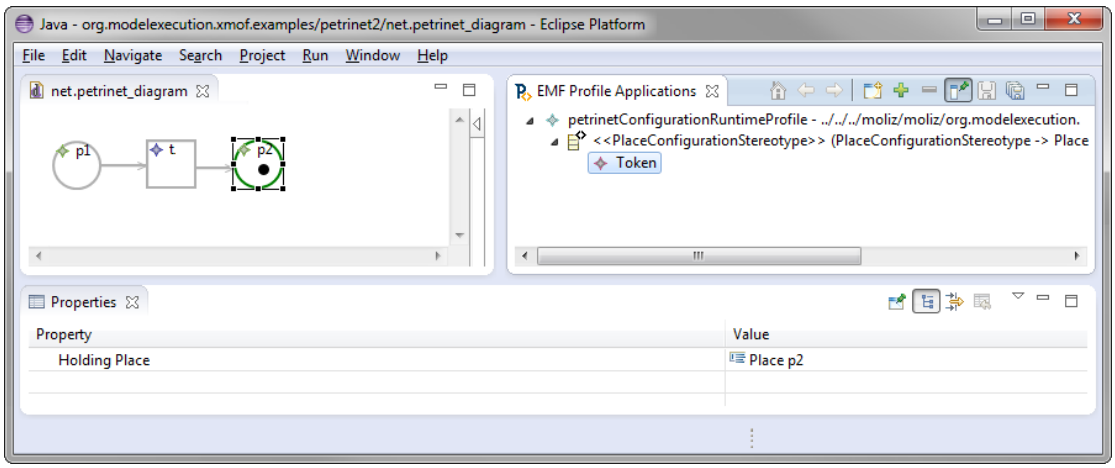


Figure 5.15: Screenshot of model execution tool

Stereotype as well as the Token instance contained by this stereotype application. The *properties view* shown at the bottom displays the values of this Token instance, which is the value of the reference holdingPlace referring to the place *p2*.

5.5.2 Model Debugging

As the presented model execution tool, also the model debugger is implemented as an Eclipse plugin. In particular, the model debugger is integrated with the Eclipse Debug Framework[4], which provides language independent facilities and mechanism for debugging, such as program launching, event notifications, and debug user interface components.

[4] `http://www.eclipse.org/eclipse/debug`, accessed 10.07.2014

The model debugger enables the stepwise execution of models by utilizing the model execution environment presented in Section 5.4. In particular, it employs the model execution capability as well as the event mechanism and command interface offered by the model execution environment for performing model executions, suspending model executions after the completion of one execution step, as well as resuming suspended executions. The extent of one execution step is configurable for the respective modeling language. After the completion of each execution step, i.e., when a model execution is suspended, the runtime state of the model is presented to the user. Therefore, like the presented model execution tool, also the model debugger generates a runtime profile application for the debugged model and loads it into the modeling editor displaying the model.

Execution step configuration. The model execution environment for xMOF enables the execution of models by executing the xMOF-based configuration of the used modeling language for the model by means of the extended fUML execution environment presented in Chapter 4. Thereby, it provides access to the extended fUML execution environment for utilizing its event mechanism, command interface, and trace model. By using the event mechanism and command interface, it is possible to observe the runtime state of an xMOF-based configuration being executed by the fUML virtual machine and suspend it either after the completion of each execution step or at a set breakpoint. Therewith, it is possible to suspend the execution of a model at any point in time of the execution. This possibility is leveraged by the model debugger for stepwise executing models. Thereby, one execution step of a model comprises usually the execution of several execution steps of the xMOF-based configuration by the fUML virtual machine. For defining the extent of one execution step, the model debugger provides a simple configuration mechanism in terms of a *debugger configuration metamodel*, which is shown in Figure 5.16. A debugger configuration (metaclass DebuggerConfiguration) is defined for one xMOF-based configuration (reference configurationPackage) and consists of a set of step definitions (metaclass StepDefinition) defining at which location of the execution of this xMOF-based configuration the execution of a model shall be suspended. Currently, the debugger configuration metamodel allows the definition of this location in terms of activity nodes contained by the xMOF-based configuration (metaclass ActivityNodeStepDefinition) denoting that the execution of a model shall be suspend when the defined activity node is reached. However, more complex definitions of steps could be easily supported, such as OCL expressions on the state of the executing model or the underlying executing xMOF-based configuration. Besides step definitions, the debugger configuration also allows the definition of the editor that shall be used for displaying the debugged model as well as its runtime state at suspension (attribute editorID).

Example. The debugger configuration of the Petri net language consists of one activity node step definition referring to the initial node of the activity defining the behavior of the *fire()* operation of the configuration class TransitionConfiguration (cf. Figure 5.7). Thus, it defines that the execution of a Petri net model shall be suspended for the first time after the initial token distribution has been initialized but before the first transition is fired and then again before each firing of a transition.

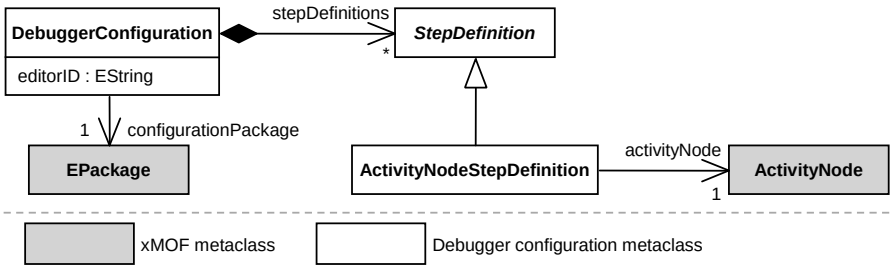


Figure 5.16: Debugger configuration metamodel

Stepwise execution. To achieve the stepwise execution of a model, the model debugger reads in the debugger configuration of the respective modeling language and adds for each activity node referenced by an activity node step definition a breakpoint using the command interface provided by the model execution environment. Thereafter, the execution of the model, i.e., the execution of the modeling language's xMOF-based configuration for the model, is started. Due to the set breakpoints, the model execution is suspended whenever an activity nodes referenced by an activity node step definitions is reached during the underlying execution of the xMOF-based configuration. At suspension, the debugger generates just as the model execution tool a runtime profile application, which captures the current runtime state of the debugged model (cf. Section 5.5.1). The generated runtime profile application is then loaded into the modeling editor displaying the debugged model, enabling the user to inspect the current runtime state of the model. Furthermore, the current location of the suspended model, that is the current location of the suspended underlying executing xMOF-based configuration, is displayed by the model debugger. This information about the current location of the suspended execution can be easily retrieved from the trace model captured by the model execution environment for the executing xMOF-based configuration. For resuming or terminating the model execution, the user can make use of debug commands provided by the Eclipse debug framework and implemented by our model debugger.

Example. Figure 5.17 shows a screenshot of the model debugger for stepwise executing the example Petri net model (cf. Figure 5.8). Thereby, the first execution step of the Petri net model, i.e., the initialization of the initial token distribution, has been completed and the execution has been suspended. The *debug view* depicted on the top left-hand side shows the suspension location of the Petri net model, that is the initial node contained by the activity defined for the operation fire() of the configuration class TransitionConfiguration. The operation fire() has been called by the operation run() of the configuration class NetConfiguration that in turn has been called by the operation main() of the same configuration class. As shown in the *variables view* depicted on the top right-hand side, the operation fire() is currently being executed for the transition *t*. The model is displayed by the graphical editor of the Petri net language depicted on the middle left-hand side and the generated runtime profile application capturing the current runtime state of the model is loaded into the editor. As can be seen in the *EMF Profile applications view* on the middle right-hand side, one token resides currently at place *p1*.

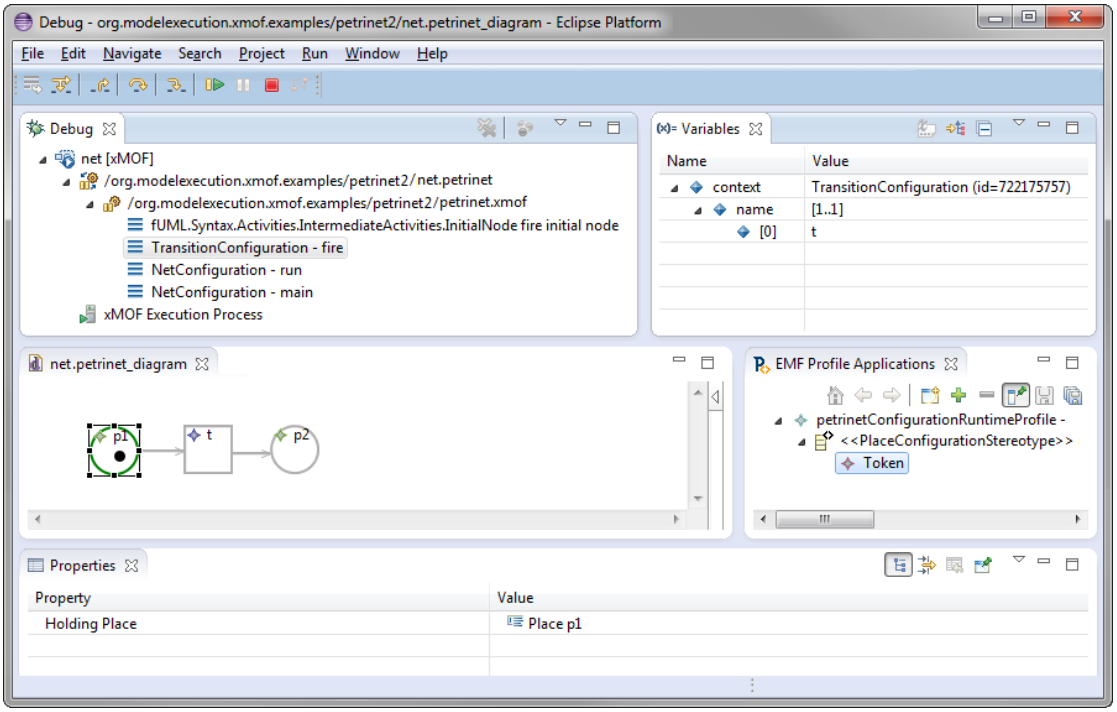


Figure 5.17: Screenshot of model debugger

5.5.3 Discussion on Semantics-based Tool Development

We have shown how the model execution environment for xMOF can be utilized to efficiently develop semantics-based tools for executable modeling languages defined with fUML. In particular, we have shown how the runtime state of a model can be obtained from the model execution environment during or after the execution, as well as how the execution of a model can be observed, controlled, and analyzed. Therefore, we implemented a model execution tool as well as a model debugger on top of the model execution environment that provide model execution and model debugging capabilities, respectively, based on the event mechanism, command interface, and trace model offered by the extended fUML execution environment underlying the model execution environment for xMOF.

Table 5.1 shows which extensions of the fUML execution environment have been applied for implementing the respective tool. Both tools make use of the execution environment access provided by the command interface for retrieving the runtime state of a model after or during its execution. By observing the trace events issued by the event mechanism during the execution of a model as well as by utilizing the execution control capabilities offered by the command interface, the implemented model debugger is enabled to stepwise execute a model. Furthermore, the issued trace events as well as the trace model capturing the executed parts of the model and their inputs and outputs enable the model debugger to retrieve the location of a suspended model execution in terms of operations being currently executed as well as their context objects.

	Event mechanism	Command interface	Trace model
Model execution			
Retrieval of final runtime state	-	Execution environment access	-
Model debugging			
Stepwise execution	Trace events	Execution control	-
Retrieval of execution location and context objects	Trace events	-	Executions Inputs and outputs
Retrieval of runtime state	-	Execution environment access	-
Semantic model differencing			
Construction of trace for differencing	Trace events Extensional value events	-	-

Table 5.1: Applications of fUML execution environment extensions for developing semantics-based tools

In Chapter 6, we will present another analysis tool built on top of the model execution environment, namely a generic framework for realizing semantic model differencing operators. This framework enables comparing two models semantically, i.e., based on their behavior, by comparing their trace models capturing the detailed runtime behavior of the two models. For facilitating the comparison of trace models, a simplified trace is constructed for the two compared models by utilizing the trace events and extensional value events issued by the event mechanism provided by the model execution environment (cf. Table 5.1).

By leveraging the event mechanism, command interface, and trace model provided by the model execution environment for xMOF, further analysis tools building upon the executability and, thereby, on the behavioral semantics of a modeling language can be implemented, such as testing environments and dynamic model analysis tools. Thereby, such tools operate on the behavioral semantics specification of a modeling language to provide analysis capabilities for models created with the modeling language. By observing, controlling, and analyzing the execution of the xMOF-based configuration of the used modeling language, the execution of a model can be observed, controlled, and analyzed. This capability builds the basis for implementing analysis method and techniques for executable modeling languages defined with fUML.

5.6 Summary

Due to the lack of a commonly accepted or even standardized language for formally defining the behavioral semantics of modeling languages, tools for executing models and analyzing the models' execution behavior for comprehension, exploration, validation, or verification purposes have to be implemented manually for the respective modeling language. To overcome this issue, we propose to apply fUML as a semantics specification language in MDE for defining executable modeling languages. Therefore, we presented a strategy for integrating fUML—in particular its activity modeling concepts and action language—with existing metamodeling languages. Furthermore, we elaborated a methodology for developing semantics specifications with fUML that integrates seamlessly with existing metamodeling methodologies and metamodeling environments. By leveraging fUML's execution environment, models conforming to a modeling language whose behavioral semantics is defined with fUML can be executed. Required pre- and post-processing steps are performed by the generic model execution environment that we built on top of fUML's execution environment. Thanks to our extensions incorporated into fUML's execution environment comprising an event mechanism, command interface, and trace model, it is possible to observe, control, and analyze model executions. Thus, by leveraging these extensions, semantics-based tools, such as model debuggers, model testing environments, and dynamic model analysis tools, can be implemented in a more efficient way. With the integration of fUML into existing metamodeling languages, metamodeling methodologies, and metamodeling environments, we aimed at providing a stimulus towards the establishment of fUML as common semantics specification language in MDE, as well as towards the automation of the development of semantics-based tools. In the following, we provide a brief summary of the contributions presented in this chapter.

Semantics specification language. By investigating distinct strategies for using fUML as a semantics specification language, we selected the integration-based operational strategy due to the more direct way of specifying the behavioral semantics as well as the better integration with existing metamodeling methodologies and metamodeling environments. For realizing this strategy, the reflexive definition of metamodeling languages as well as their support of inheritance can be exploited, which allows the integration of fUML into the respective metamodeling language without having to modify its meta-metamodel. This has been demonstrated on the metamodeling language Ecore. The resulting language is an executable metamodeling language enabling to define not only the abstract syntax of a modeling language but also the behavioral semantics of a modeling language in an operational way using fUML.

Semantics specification methodology. To enable a systematic and efficient development of behavioral semantics specifications with fUML, we elaborated a methodology comprising a set of processes, techniques, and supporting tools. The defined processes are orthogonal to existing metamodeling methodologies and only extend them concerning the specification of the behavioral semantics of modeling languages. The provided techniques and supporting tools can be integrated into existing metamodeling environments for supporting the development of behavioral semantics specifications as well as the execution of models based on these specifica-

tions. This seamless integration with existing metamodeling methodologies and metamodeling environments is achieved by a clear separation of the modeling language's behavioral semantics definition from its abstract syntax definition. We provide tool support for the elaborated methodology for the metamodeling environment EMF.

Model execution environment. To enable the execution of models, we developed a model execution environment on top of fUML's execution environment. This model execution environment takes as input the model to be executed as well as the fUML-based behavioral semantics specification of the respectively used modeling language. These two artifacts are converted to fUML and provided as input to the fUML virtual machine, which carries out the execution. In particular, the behavioral semantics specification is converted into an fUML model and the model to be executed is converted into fUML extensional values. The obtained fUML model is then executed for the obtained fUML extensional values by means of the fUML virtual machine. Thereby, the model execution environment provides access to the runtime state of the executing model as well as to the event mechanism, command interface, and trace model incorporated into fUML's execution environment.

Semantics-based tool development. The model execution environment provides the basis for developing semantics-based tools that build upon the executability of a modeling language in an efficient way, as it provides on the one hand the capability to directly execute models based on the fUML-based behavioral semantics specification of the used modeling language, and on the other hand means for observing the state of the execution, controlling the execution, and analyzing the execution. Therefore, semantics-based tools utilize the event mechanism, command interface, and trace model incorporated into fUML's execution environment and made accessible by the model execution environment. We have shown how this is possible by presenting the implementation of a model execution tool and a model debugger realized on top of the developed model execution environment.

We evaluated the adequacy of fUML and our semantics specification methodology for developing executable modeling languages by conducting case studies in which we applied them to selected modeling languages. The case studies and evaluation results are presented in Section 7.2.

5.7 Related Work

In Section 2.2.2, we provided an overview of existing approaches for formally defining the behavioral semantics of modeling languages. Thereby we distinguished between translational semantics approaches and operational semantics approaches. In the following, we discuss two existing semantics specification approaches in more detail, namely Kermeta [72,107] and DMM [41, 64, 145]. These two approaches are highly related to our fUML-based semantics specification approach, as they are like our approach operational semantics approaches. Furthermore, we regard them as being among the most advanced and active semantics specification approaches developed and applied in MDE. In the discussion of these approaches, we provide insights into

how semantics specifications are developed using the respective approach, how models are executed based on developed semantics specifications, as well as which facilities are provided for developing semantics-based tools on top of semantics specifications.

Besides Kermeta and DMM, we also briefly discuss work done by Lai and Carpenter [82, 83], who also propose to use fUML as semantics specification language, but focus on static verification of semantics specifications developed with fUML.

Kermeta

Kermeta [72, 107] is a metamodeling language that provides metamodeling concepts for defining not only the abstract syntax but also the behavioral semantics of modeling languages. The metamodeling concepts provided by Kermeta for defining the abstract syntax of modeling languages are compliant with EMOF—a subset of MOF—as well as Ecore. For defining the behavioral semantics of modeling languages, Kermeta provides an imperative and object-oriented action language. One key characteristic of Kermeta is that it is an aspect-orientated language implementing the open-class mechanism that enables the extension of existing metaclasses with additional attributes, references, and operations. Using this aspect weaving capabilities, Kermeta enables the extension of the metamodel of a modeling language with the definition of the modeling language's behavioral semantics. Besides defining modeling languages, Kermeta can also be used to develop any kind of tool for processing models, as it enables loading, modifying, and saving models. Kermeta is integrated with EMF and provides a set of Eclipse plugins that support in using Kermeta for defining modeling languages and developing modeling language tool support. These plugins include a textual editor for creating Kermeta programs, a run configuration for executing Kermeta programs, and a converter between Kermeta programs and Ecore-based metamodels.

Semantics specification. Kermeta enables the definition of the behavioral semantics of modeling languages in an operational way. Therefore, aspects are defined for weaving the definition of the behavioral semantics into a modeling language's metamodel. The definition of the behavioral semantics consists of the definition of runtime concepts enabling the capture of the runtime state of an executing model, as well as the definition of the computational steps that are involved in executing a model and change an executing model's state.

For defining the runtime concepts capturing a model's runtime state, aspects are defined that introduce additional attributes, references, and metaclasses into a modeling language's metamodel. Introduced attributes, references, and metaclasses correspond to the structural information captured in xMOF-based configurations.

For defining the computational steps of executing a model, aspects are defined that introduced operations into existing metaclasses. While in xMOF, the behavior of operations is defined using fUML conform activities, Kermeta defines its own imperative action language, which provides block statements, conditional statements, loop statements, local variable declarations, call expressions, assignment expressions, literal expressions, lambda expressions, exception handling, as well as OCL-like expressions, such as the collection operations each, forAll, and select. Furthermore, it is possible do define and check class invariants as well as operation preconditions and postconditions.

```
1  aspect class Transition
2  {
3    operation fire() is do
4      self.output.each{ o | o.addToken()}
5      self.input.each{ i | i.removeToken()}
6    end
7
8    operation isEnabled() : Boolean is do
9      result := input.select{ i | i.heldTokens.size == 0}.isEmpty()
10   end
11 }
12
13 aspect class Place
14 {
15   attribute heldTokens : Set<Token>
16
17   operation addToken() is do
18     var newToken : Token init Token.new
19     newToken.holdingPlace := self
20     self.heldTokens.add(newToken)
21   end
22
23   operation removeToken() is do
24     var removedToken : Token init self.heldTokens.iterator().next()
25     removedToken.holdingPlace := void
26     self.heldTokens.remove(removedToken)
27   end
28 }
```

Listing 5.1: Semantics specification of Petri nets with Kermeta (excerpt)

Listing 5.1 shows an excerpt of the semantics specification of our Petri net language defined with Kermeta 1.4.1, which is equivalent to the semantics specification defined with xMOF (cf. Section 5.3). The shown aspects Transition and Place correspond to our configuration classes TransitionConfiguration and PlaceConfiguration, respectively. Hence, they define the operations fire(), isEnabled(), addToken(), and removeToken(), as well as the reference heldTokens. For instance, the operation fire() defined in line 3, calls the operation addToken() for each output place of the transition as well as the operation removeToken() for each input place of the transition. Therefore, the OCL-like collection operation each provided by Kermeta is used.

Model execution. For enabling the execution of models with Kermeta, an additional so-called *interpreter class* has to be introduced into the semantics specification of the modeling language. This interpreter class has to define the main operation of the semantics specification, which is responsible for loading the model to be executing and starting the execution process by calling operations introduced into the metaclasses of the modeling language defining their behavioral semantics. Thereby, the main operation can take String values as input, for instance to receive the file path of the model to be executed. In version 1 Kermeta programs are executed by a Java-based interpreter. In version 2, Kermeta programs are compiled into Scala programs for execution.

Semantics-based tool development. Kermeta can not only be used to define modeling languages, but it can also be used as a programming language for implementing any kind of tool for processing models. In particular, semantics-based tools can be developed that build upon the behavioral semantics specification of a modeling language defined with Kermeta. However, this usually requires the extension of the behavioral semantics specification with certain functionality required for implementing the respective tool. Therefore, again the aspect weaving capabilities provided by Kermeta can be utilized. For instance, to capture execution traces required for implementing dynamic analysis techniques, one could implement aspects that override certain operations defined as part of the behavioral semantics specification of a modeling language for additionally capturing trace information. Our approach differs in this respect, as semantics-based tools are developed by utilizing the event mechanism, command interface, and trace model provided by the extended fUML execution environment. Hence, the behavioral semantics specification itself neither has to be extended nor modified for implementing semantics-based tools with our approach.

Kermeta is currently undergoing a major evolution step in the course of its integration with the *K3 model-based language workbench* [35][5]. The new action language of K3 usable for defining the behavioral semantics of a modeling language is called K3AL, which is built on top of the programming language Xtend[6]. As Kermeta 1 and Kermeta 2, K3AL supports aspect-oriented programming, which enables the extension of existing metaclasses with attributes, references, and operations. For defining the behavior of operations using K3AL, Xbase [40] is used.

Dynamic Meta Modeling (DMM)

DMM [41, 64, 145] is an approach for defining the behavioral semantics of modeling languages in an operational way using graph transformation rules. Therefore, DMM provides its own graph transformation language that enables the expression of manipulations of models in a declarative way. Advanced features provided by this graph transformation language are negative application conditions, universal quantified structures, and rule invocations. The concrete syntax of DMM graph transformation rules is inspired by the graphical notation of UML communication diagrams and merges the left-hand side and right-hand side graph of a graph transformation rule into a single graph. Due to the mathematical foundations of graph transformations underlying DMM, behavioral semantics specifications defined with DMM can be formally analyzed for proving properties of the semantics specification of a modeling language itself or of models conforming to the respective modeling language. DMM is integrated with Ecore and, hence, enables the definition of the behavioral semantics of modeling languages whose abstract syntax is defined by means of Ecore-based metamodels.

Semantics specification. A semantics specification developed with DMM consists of three parts. The first part is the so-called *runtime metamodel*, which is an Ecore-based metamodel defining the runtime concepts needed for expressing the state of executing models. The second

[5] `http://github.com/diverse-project/k3/wiki`, accessed 14.08.2014
[6] `http://www.eclipse.org/xtend`, accessed 14.08.2014

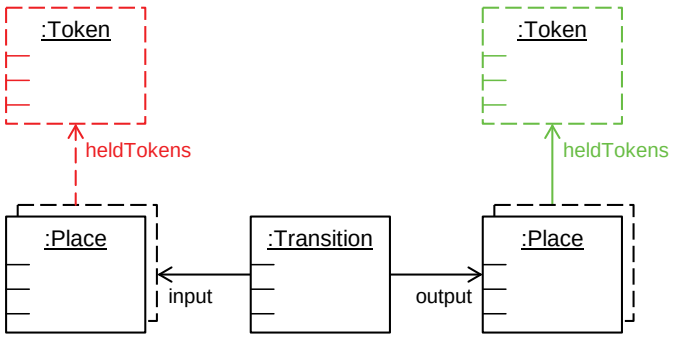


Figure 5.18: Semantics specification of Petri nets with DMM (excerpt)

part of a DMM semantics specification is the *runtime transformation*, which transforms models to be executed into *runtime models* being instances of the runtime metamodel. The third part of a DMM semantics specification is the *DMM rule set*, which defines the actual behavioral semantics of the modeling language, i.e., the computational steps involved in executing a model, using graph transformation rules that operate on the runtime metamodel. The runtime transformation, as well as the DMM rule set are defined using DMM's graph transformation language.

While a DMM runtime metamodel corresponds to the structural part of an xMOF-based configuration, i.e., configuration classes as well as their attributes and references, the DMM rule set corresponds to operations introduced into configuration classes as well as their behavior definitions in terms of fUML conform activities. Both DMM and xMOF require for the execution of a model, that this model is first transformed, namely into an instance of the runtime metamodel in the case of DMM and into an instance of the xMOF-based configuration in the case of xMOF. In xMOF, this transformation is generic due to the convention that an xMOF-based configuration contains one configuration class for each metaclass. In contrast, no such convention is defined by DMM resulting in the need for explicitly defining a transformation of models into runtime metamodel instances. However, DMM provides a technique for automatically generating an initial runtime transformation that can be adapted by the language designer.

Figure 5.18 shows the DMM rule defining the behavioral semantics of our Petri net language. This rule is applied to a Petri net if all input places of a contained transition hold at least one token. On the application of this rule, one token is removed from each input place (indicated by the red color of the left Token node), and one token is added to each output place of the transition (indicated by the green color of the right Token node). For expressing this, the rule makes use of universal quantified structures. In particular, the Place nodes are quantified with 1..* (indicated by the multi-object notation, where the top rectangle has solid borders) and the Token nodes are quantified as nested (indicated by the dashed borders) meaning that they are matched in conjunction with the Place node they are connected to.

Model execution. For executing models according to a semantics specification defined with DMM, the model to be executed as well as the runtime metamodel and the DMM rule set of the semantics specification are translated into a GROOVE grammar [132]. In particular, the runtime metamodel is translated into a type graph, the DMM rule set is translated into a set of

GROOVE graph transformation rules, and the runtime model obtained by applying the runtime transformation to the model to be executed is translated into a host graph. The GROOVE tool set[7] is then utilized for performing the model execution by applying the obtained GROOVE graph transformation rules to the obtained host graph. The result of the execution is a labeled transition system capturing each possible state of the executed models as well as transitions between them caused by rule applications. This labeled transition system can then be used to perform formal analyses of the executed model. For this purpose the GROOVE tool set provides a model checker that enables the checking of properties expressed in temporal logic.

Semantics-based tool development. For developing semantics-based tools on top of semantics specification created with DMM, the GROOVE tool set can be utilized, which acts as underlying execution environment of DMM. In particular, the GROOVE tool set can be utilized for generating the state space of models, i.e., labeled transition systems, in a controlled way, as well as for checking properties on generated state spaces expressed in temporal logic formulas. By doing so, semantics-based tools have been implemented on top of DMM that work generically for any modeling language whose behavioral semantics if formally defined with DMM graph transformation rules. In particular, researchers and students associated with the research group developing DMM have implemented a testing environment for testing DMM semantics specifications based on the execution of example models [146], a formal analysis tool for verifying functional requirements on executable models [147], a non-functional analysis tool for evaluating performance properties of executable models [145, pp. 195–209], as well as a model debugger [8].

Developing semantics-based tools on top of DMM-based semantics specifications of modeling languages requires utilizing the GROOVE tool set as the underlying model execution infrastructure, handling GROOVE grammars representing the definition of a modeling language and conforming models, as well as handling and analyzing labeled transition systems. In contrast, with our fUML-based semantics specification approach, semantics-based tools can be implemented by utilizing the extended fUML execution environment that enables the observation, control, and analysis of the runtime behavior of executing models directly. Hence, with our approach, tool developers do not need to cope with an additional language or framework, but only with fUML and its execution environment.

Static Verification of fUML-based Semantics Specifications

We presented our proposal of using fUML as semantics specification language at the Domain-Specific Modeling (DSM) workshop held in October 2012 [98]. Only three months earlier, in July 2012, Lai and Carpenter presented the very same proposal at the workshop on Behavioural Modelling Foundations and Application (BM-FA) [82]. While both proposals suggest the usage of fUML for defining the behavioral semantics of modeling languages in an operational way, they differ in the proposed integration of fUML with existing metamodeling languages.

As discussed in Section 5.2.1, there are two alternatives for realizing the operational semantics approach for fUML. We chose the integration-based approach that integrates the behavioral

[7] `http://groove.cs.utwente.nl`, accessed 14.08.2014

part of fUML, i.e., fUML's activity modeling concepts as well as action language, with existing metamodeling languages resulting in executable metamodeling languages (cf. Section 5.2.2). In contrast, Lai and Carpenter chose the transformation-based approach [82, 83]. In their approach, the abstract syntax as well as the behavioral semantics of a modeling language is defined in terms of an fUML model. Thereby, the abstract syntax is defined with fUML classes, and the behavioral semantics is defined with operations introduced in these fUML classes. The behavior of these introduced operations is defined in terms of fUML activities, which are expressed either using fUML directly or using Alf. From the defined fUML classes, an Ecore-based metamodel is automatically generated that can be used to create models conforming to the modeling language.

Lai and Carpenter focus on the static verification of modeling language definitions created with fUML and Alf to identify structural defects, such as syntax errors and inconsistencies. For detecting runtime defects, the authors propose a testing approach. However, they do not utilize the standardized fUML execution environment for executing models and testing modeling language definitions, but propose to generate Java code from fUML-based modeling language definitions and apply testing techniques available for Java [83]. In contrast, the aim of our work is to establish the fUML execution environment as foundation for efficiently developing semantics-based tools that build upon the executability of models provided by a formal definition of the used modeling language's behavioral semantics with fUML.

CHAPTER 6

Semantic Model Differencing

6.1 Design Rationale

As models constitute the main software artifacts in MDE, managing their evolution constitutes a key concern in MDE. One important technique in this realm is *model differencing*, which is concerned with identifying differences among independently developed or consecutive versions of models.

Significant advances in model differencing have been made in the past years by the proposal of a variety of model differencing approaches. The majority of these existing model differencing approaches compare models based on their abstract syntax representation. The differencing algorithm first identifies corresponding model elements among the two models to be compared and then performs a fine-grained comparison of all corresponding model elements. This results in a set of *syntactic differences* among the models, such as model elements that only exist in one of the two compared models and modifications of the attribute values of corresponding model elements that exist in both models. The identified syntactic differences among the compared models are usually represented in terms of edit operations, such as add, delete, and update operations. As shown by Alanen and Porres [3] and later by Lin *et al.* [90], syntactic differencing algorithms can be designed in a generic manner by incorporating the modeling language's metamodel into the algorithm for reasoning about the structure of the models to be compared. This means, that syntactic differencing algorithms can be applied to models conforming to any modeling language as they operate on the used modeling language's abstract syntax definition for identifying syntactic differences among models.

Syntactic differences among models constitute valuable and efficiently processable information sufficient for several application domains. However, they can only approximate *semantic differences* among models with respect to the models' meaning [63]. As pointed out by Maoz *et al.* [94], a few syntactic differences among models may induce very different semantics and syntactically different models may still induce the same semantics. Semantic model differencing enables several additional analyses compared to syntactic differencing, such as the verification of the semantic preservation of changes like refactorings and the identification of se-

mantic conflicts among concurrent changes. Moreover, the identification of semantic differences among models provides the basis for comprehending the evolution of models, as it enables reasoning about the meaning of a change, that is the impact a syntactic change of a model has on the model's semantics. Hence, being able to reveal semantic differences among models constitutes a crucial basis for supporting collaborative work on models as well as for carrying out model management activities, such as model versioning and refactoring, which can be supported by automated analyses of revealed semantic differences for identifying semantic conflicting changes and causes of semantic differences.

Significant advances towards *semantic model differencing* have been recently achieved by Maoz *et al.* [94]. They propose an approach for defining enumerative *semantic differencing operators* yielding so-called *diff witnesses*, which constitute semantic interpretations over a model that are valid for only one of two compared models. In this approach, two models to be compared are translated into an adequate semantic domain whereupon dedicated algorithms are applied to calculate semantic differences in terms of diff witnesses. Defining a semantic differencing operator for a modeling language following this approach requires the implementation of a translation of models into an adequate semantic domain, a dedicated analysis algorithm within this semantic domain for computing semantic differences, and a translation of the obtained analysis result back to the originally used modeling language in the form of diff witnesses. Following this procedure, Maoz *et al.* defined dedicated semantic differencing operators for UML activity diagrams and UML class diagrams called ADDiff [95] and CDDiff [96], respectively. Developing semantic differencing operators for a specific modeling language in this way, however, still remains a major challenge, as one has to develop often non-trivial transformations encoding the semantics of the modeling language into a semantic domain, perform analyses dedicated to semantic differencing in this semantic domain, and translate the results into diff witnesses and into the originally used modeling language. Notably, this challenging process has to be repeated for every modeling language.

However, given that the behavioral semantics of a modeling language is defined explicitly and formally, the idea of generic syntactic model differencing [3, 90] based on a modeling language's abstract syntax definition can be transferred to realize *generic semantic model differencing* based on a modeling language's behavioral semantics definition. Moreover, if an operational semantics approach is applied for defining the behavioral semantics of a modeling language as advocated in Chapter 5, it is possible to reason about semantic differences among models directly in the used modeling language, i.e., without the involvement of a different language like in the approach proposed by Maoz *et al.* Realizing such a generic semantic model differencing approach requires techniques to extract *semantic interpretations* of the models to be compared from the behavioral semantics specification of the used modeling language and to compare these semantic interpretations for identifying semantic differences among the models.

Following this idea, we proposed a *generic framework* that enables the realization of *semantic model differencing operators* for specific modeling languages [85, 86]. According to the idea of generic syntactic model differencing, we propose to utilize the behavioral semantics specification of a modeling language for supporting semantic model differencing. Thereby, semantic interpretations of the models to be compared, which are required for performing semantic differencing, are extracted from the behavioral semantics specification by leveraging the executability

of the models provided by the behavioral semantics specification. In particular, the behavioral semantics specification is utilized to execute the models to be compared and obtain *execution traces* constituting semantic interpretations of the models. By comparing these execution traces, semantic differences among the models can be identified. For comparing execution traces, syntactic model differencing approaches may be employed, in particular, by defining dedicated *match rules* that indicate which syntactic differences among the execution traces constitute semantic differences among the models. Thereby, execution traces leading to the identification of semantic differences constitute *diff witnesses*, that are manifestations of semantic differences among the models. They enable modelers to reason about a model's evolution and can be further processed for carrying out model management activities, such as model versioning.

For modeling languages whose behavioral semantics is specified using fUML as proposed in Chapter 5, it is possible to obtain execution traces in terms of trace models by executing models using the extended fUML execution environment as presented in Section 5.4. While these trace models are tailored to *fUML-based behavioral semantics specifications*, it is in general possible to obtain execution traces from *operationally defined behavioral semantics specifications*. Thus, generic semantic model differencing can be realized based on operationally defined behavioral semantics specifications in general.

As the semantic model differencing is performed based on *concrete execution traces* of two compared models, semantic differences identified by analyzing them depend on the *concrete inputs* processed by the models during their execution. Thus, concrete inputs relevant to the semantic model differencing have to be provided as input to the semantic model differencing. To fully automated semantic model differencing, these inputs have to be automatically generated. This is possible by analyzing the behavioral semantics specification of the used modeling language for the models to be compared.

The characteristics of our semantic model differencing framework are summarized in the following.

Generic with respect to modeling language. Our semantic model differencing framework enables the realization of semantic model differencing operators for any executable modeling language whose behavioral semantics is explicitly and formally defined.

Generic with respect to semantics specification approach. The semantic model differencing framework can be instantiated for any operational semantics approach that enables executing models and obtaining execution traces based on which the semantic differencing is performed.

Configurable with respect to semantic equivalence criteria. The semantic model differencing is performed by syntactically comparing execution traces of models according to match rules. These match rules may be tailored to the semantic equivalence criterion relevant to the usage scenario of the respective modeling language.

Automatable with respect to relevant inputs. Inputs for compared models relevant to the semantic model differencing may be either defined manually or generated from behavioral semantics specifications.

In this chapter, we present our generic semantic model differencing framework as described in the following. In Section 6.2, we present an overview of our generic semantic model differencing framework. Subsequently, we show how the framework can be instantiated for fUML-based semantics specifications and how it can be instantiated for operational semantics approaches in general in Section 6.3 and Section 6.4, respectively. In Section 6.5, we discuss how inputs required for fully automating semantic model differencing can be generated from fUML-based behavioral semantics specifications.

6.2 Overview of the Semantic Model Differencing Framework

We propose a *generic* semantic model differencing framework that can be instantiated to realize semantic model differencing operators for specific modeling languages. The framework utilizes the *behavioral semantics specification* of a modeling language, which can be defined using existing semantics specification approaches, such as xMOF introduced in Chapter 5, Kermeta [72, 107], or DMM [41, 64, 145]. Behavioral semantics specifications can be used for various application domains including, for instance, model simulation, verification, and validation. We use behavioral semantics specifications also for identifying semantic differences among models. For doing so, we exploit the fact that behavioral semantics specifications enable the execution of models and that the identification of semantic differences among models is possible based on *execution traces*, as they reflect the models' behavior and, hence, constitute semantic interpretations of the models. Our framework is applicable irrespective of how the examined models are executed—e.g., through an interpreter, code generation, or the translation to another language—but only requires that execution traces reflecting the models' behavior and, therewith, their semantics are captured during the execution of the models.

Figure 6.1 depicts an overview of our semantic model differencing framework. For identifying semantic differences among two models M_1 and M_2, three steps are performed by the framework, namely syntactic matching, model execution, and semantic matching. These steps are described in more detail in the following.

1. Syntactic matching. In the *syntactic matching* step, syntactically corresponding elements of the two compared models M_1 and M_2 are identified based on defined syntactic match rules $MatchRulesSyn$. Thereby, syntactic correspondences $C^{syn}_{M_1,M_2}$ between the two models are established. This step is realized using existing syntactic model differencing approaches.

2. Model execution. In the *model execution* step, the models M_1 and M_2 are executed for inputs I_{M_1} and I_{M_2} relevant to the semantic model differencing based on the behavioral semantics specification of the modeling language the models conform to. During the model execution, the execution traces T_{M_1} and T_{M_2} are captured, which constitute semantic interpretations of the executed models. The execution of the models as well as the capturing of execution traces is realized by utilizing the execution environment provided by the respectively employed semantics specification approach.

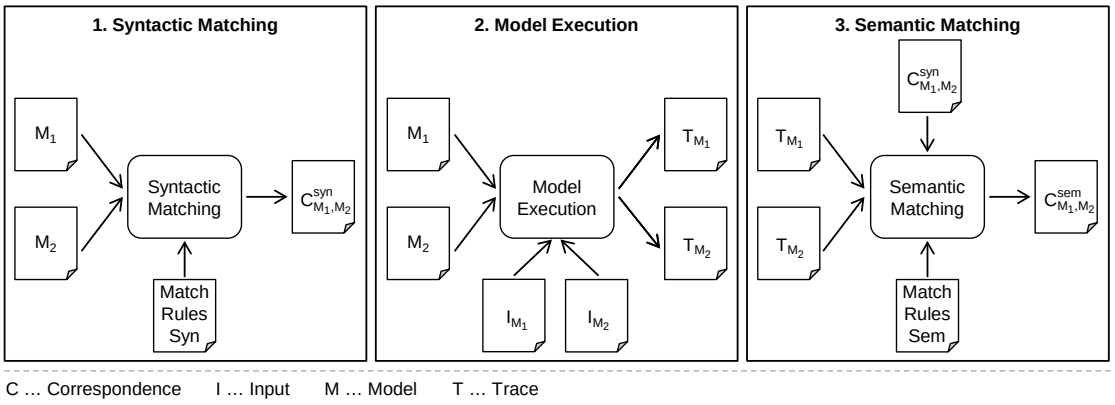


Figure 6.1: Overview of semantic model differencing framework

Our framework is based on the following two assumptions. First, it is assumed that the model execution is deterministic, meaning that the execution of a model yields for a given input always the same execution trace. Second, it is assumed that execution traces are of finite size.

3. Semantic matching. In the *semantic matching* step, the captured execution traces T_{M_1} and T_{M_2} are compared based on semantic match rules $MatchRulesSem$ to establish semantic correspondences $C^{sem}_{M_1,M_2}$ between the models. The semantic match rules define the semantic equivalence criterion to be applied for the semantic differencing. This means, that they define which syntactic differences among the execution traces T_{M_1} and T_{M_2} constitute semantic differences among the two compared models. Thereby, two models are semantically equivalent, if the traces match according to the semantic match rules. Traces, which do not match constitute diff witnesses that are manifestations of semantic differences among the compared models. In the semantic matching, also the syntactic correspondences between the compared models $C^{syn}_{M_1,M_2}$ may be taken into account. Also this step is realized by applying existing syntactic model differencing approaches for comparing execution traces.

Our semantic model differencing framework is *generic*, because it enables the implementation of semantic differencing operators for *any modeling languages* whose behavioral semantics is defined using *any operational semantics approach* that enables executing models and obtaining execution traces. Thereby, it follows the spirit of generic syntactic differencing—that is utilizing metamodels for obtaining the information on the syntactic structure of models required for performing syntactic differencing—by utilizing behavioral semantics specifications for obtaining the information on the semantics of models for performing the semantic differencing. From all artifacts involved in the semantic differencing, only the semantic match rules are specific to the realization of a semantic differencing operator for a modeling language. Thereby, the framework is *configurable* regarding the semantic equivalence criterion to be applied, as it allows the realization of distinct semantic differencing operators for the same modeling language by defining distinct semantic match rules realizing the respective semantic equivalence criterion. Both

of these characteristics differentiate our semantic model differencing approach from currently existing semantic model differencing approaches.

While this section gave an overview of our semantic model differencing framework, we will discuss more insights in the following two sections. In Section 6.3, we show how the framework can be instantiated for realizing semantic model differencing operators for modeling languages whose behavioral semantics is defined using fUML as presented in Chapter 5. Thereafter, in Section 6.4, we show that an instantiation of the framework is possible for any operational semantics approach that provides the possibility to execute models and obtain execution traces for performed model executions.

6.3 Semantic Differencing for fUML-based Semantics Specifications

In Chapter 5, we have presented an operational semantics approach based on fUML that enables the definition of the behavioral semantics of modeling languages in an explicit, formal, and model-based way. In this approach, the behavioral semantics of a modeling language is defined by specifying fUML conforming activities that define an interpreter for models conforming to the modeling language. Behavioral semantics specifications defined using this approach enable the execution of models conforming to the respective modeling languages by utilizing fUML's execution environment. Therefore, we provide a generic model execution environment that is built on top of fUML's execution environment as well as on our extensions of this execution environment presented in Chapter 4. One of these extensions is concerned with providing the means for dynamically analyzing models. In particular, this extension captures trace models for model executions carried out by means of fUML's execution environment, which constitute execution traces that represent the runtime behavior of executed models. As captured trace models represent the behavior of executed models, they constitute semantic interpretations of these models. Hence, trace models captured by the extended fUML execution environment can be directly used for performing semantic model differencing. In particular, by syntactically comparing the trace models captured for two models, semantic differences among these models can be identified. The syntactic comparison of trace models can be realized by applying generic syntactic model differencing approaches that allow the definition of comparison algorithms for models conforming to arbitrary modeling languages.

In the following, we discuss the instantiation of our semantic model differencing framework for fUML-based semantics specifications in more detail. Therefore, we make again use of our Petri net language, whose abstract syntax and behavioral semantics is defined in Section 5.3.

Example. For Petri nets, different equivalence criteria exist, such as *marking equivalence*, *trace equivalence*, and *bisimulation equivalence* [46]. Using our semantic model differencing framework, we will develop a semantic model differencing operator that compares two Petri net models according to the equivalence criterion *final marking equivalence*, which we defined for illustration purposes. Two Petri net models with the same set of places are final marking equivalent, if for the same initial markings the same final markings are reached, that is if for the same initial token distribution both Petri nets reach the same final token distribution at the end of

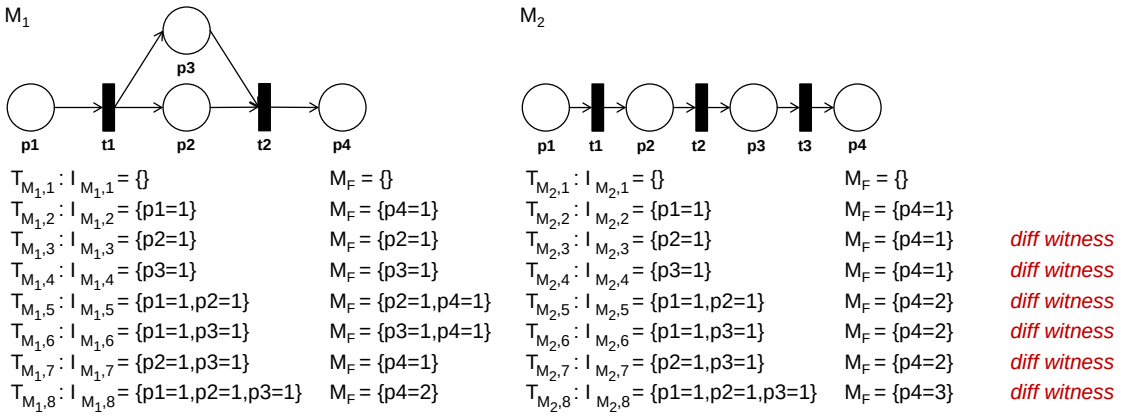


Figure 6.2: Semantic model differencing example: Models and inputs

their execution terminates. Please note that we restrict ourselves in this example to terminating Petri nets.

Figure 6.2 depicts two example Petri net models M_1 and M_2, which have the same set of places $\{p1, p2, p3, p4\}$ but comprise different transitions. Below the models, the final markings M_F of the Petri nets for given inputs I_{M_1} and I_{M_2} defining the initial markings of the Petri nets are shown. For the first input $I_{M_1,1}$ and $I_{M_2,1}$ consisting of no tokens, both Petri nets reach a final marking consisting also of no tokens. For the second input $I_{M_1,2}$ and $I_{M_2,2}$ defining an initial marking comprising one token residing at place *p1*, both Petri nets reach the same final marking comprising one token residing at place *p4*. For all other given inputs, the two Petri nets reach different final markings. Thus, the two Petri net models are not final marking equivalent and the execution traces, which we obtain by executing the models on the inputs $I_{M_1,3}$ to $I_{M_1,8}$ and $I_{M_2,3}$ to $I_{M_2,8}$ constitute diff witnesses.

1. Syntactic matching. In the syntactic matching, syntactically corresponding elements of two compared models are identified based on syntactic match rules. Therefore, existing generic syntactic model differencing approaches can be applied that allow the definition of syntactic match rules for models conforming to arbitrary modeling languages. These syntactic match rules define comparison algorithms that operate on the metamodel of the respective modeling language and specify how models conforming to this modeling language are compared to identify syntactically corresponding model elements.

For defining syntactic match rules, the implementation of our semantic model differencing framework uses the model comparison language ECL [78, 79]. ECL enables the specification of model comparison algorithms with declarative match rules that identify pairs of corresponding elements between models. These rules can be executed for comparing two models resulting in a match trace. A match trace consists of a number of matches that reference the model elements that have been compared and indicate whether these two model elements do match, i.e., correspond to each other, or do not match according to an applied match rule.

```
1  rule MatchNet
2     match left : Left!Net with right : Right!Net {
3        compare : left.places.matches(right.places) and left.transitions.matches(right.transitions)
4  }
5
6  rule MatchPlace
7     match left : Left!Place with right : Right!Place {
8        compare : left.name = right.name
9  }
10
11 rule MatchTransition
12    match left : Left!Transition with right : Right!Transition {
13       compare : left.name = right.name
14 }
```

Listing 6.1: Semantic model differencing example: Syntactic match rules for Petri nets

Example. For the Petri net language we defined three syntactic match rules with ECL, which are shown in Listing 6.1. The match rule MatchNet defines that two Net instances contained by two compared Petri net models match if both their places and transitions match. For matching places and transition, the match rules MatchPlace and MatchTransition are invoked that define that two Place instances and two Transition instances match if they have the same name.

The execution of these match rules on our example Petri net models depicted in Figure 6.2 leads to the identification of six matches, namely between the four same named places *p1*, *p2*, *p3*, and *p4* as well as between the two same named transitions *t1* and *t2*.

2. Model execution. In the model execution, the two compared models are executed for given inputs. Therefore, the generic model execution environment built on top of the extended fUML execution environment is utilized (cf. Section 5.4). For performing a model execution, the generic model execution environment executes the fUML-based behavioral semantics specification—in particular, the fUML activities contained by the behavioral semantics specifications—for the model and a given input using the extended fUML execution environment. During the model execution, the extended fUML execution environment captures a trace model representing the runtime behavior of the fUML-based behavioral semantics specification for the model and the given input. In particular, as described in Section 4.4, the trace model captures information about which parts of the fUML-based behavioral semantics specification have been executed for the model and the given input, the inputs and outputs of these executed parts, as well as token flows between them. Thus, captured trace models represent the runtime behavior of executed models for a given input and, hence, constitute semantic interpretations of the models.

For enabling the semantic differencing of two models, both models are executed for given inputs and the captured trace models are handed over to the semantic matching where they are syntactically compared for identifying semantic differences among the models.

Example. In the execution of Petri net models, first their markings are initialized according to a given input defining the initial token distribution of the Petri nets. Thereafter, enabled transitions are fired sequentially leading to new markings of the Petri nets. In Figure 6.2, eight

distinct initial markings of our example Petri nets defined by the inputs I_{M_1} and I_{M_2} as well as the resulting final markings M_F are shown.

To realize a semantic model differencing operator that applies the final marking equivalence criterion on Petri net models, we have to compare the final markings resulting from the execution of the Petri net models. This information is held by the trace models captured by the extended fUML execution environment for executed Petri net models. In particular, a trace model captures each manipulation of the model's runtime state performed during the execution in terms of value snapshots. Thus, for an executed Petri net it holds—besides other information—the information about the creation and destruction of tokens at places of the Petri net caused by the firing of enabled transitions. Using this information, we can obtain the final markings of two compared Petri net models and perform the semantic matching applying the final marking equivalence criterion.

3. Semantic matching. In the semantic matching, the trace models captured during the execution of the models to be compared are used for identifying semantic differences among these models. In particular, the trace models are syntactically compared by applying semantic match rules. These semantic match rules define which syntactic differences among the trace models constitute semantic differences among the compared models. Therefore, the semantic match rules specify a comparison algorithm that establishes correspondences between the elements contained by the trace models of two compared models. Two models are semantically equivalent, if a correspondence between the two trace models, i.e., between the single Trace instance contained by each trace model, can be established. Otherwise, i.e., if this correspondence can not be established, elements contained by the trace models, which have been compared through the application of a semantic match rule but do not correspond to each other, constitute semantic differences among the compared models. Consequently, these trace models constitute diff witnesses, testifying the existence of semantic differences among the compared models.

The semantic match rules are specific to the considered modeling language as well as to the semantic equivalence criterion to be applied. Thereby, our approach is flexible with regard to the applied semantic equivalence criterion, as semantic match rules can be expressed for different semantic equivalence criteria of the same modeling language. Depending on the usage scenario of a modeling language, different semantic equivalence criteria for models may apply. For Petri nets, for instance, we already mentioned three distinct semantic equivalence criteria, namely *marking equivalence*, *trace equivalence*, and *bisimulation equivalence* [46]. If Petri nets are, for example, used to define production processes, where the tokens residing at places represent production resources, the marking equivalence criterion might be the most suitable semantic equivalence criterion. However, if Petri nets are used to define, for example, business processes, the trace equivalence criterion might be more adequate.

Like for defining syntactic match rules, the implementation of our semantic model differencing framework makes use of ECL for defining semantic match rules.

Example. The semantic match rules for determining whether two Petri net models are final marking equivalent are shown in Listing 6.2. The rule MatchTrace is responsible for matching the trace models captured during the execution of two compared Petri net models for a given

initial token distribution. If the Petri nets are final marking equivalent for the given initial token distribution, this rule has to return true, otherwise false. Therefore, the final markings of both Petri net models are obtained from the trace models, which is represented by the set of links between tokens existing at the end of the execution and the places they reside at. If these sets of links match, the two compared Petri net models are final marking equivalent for the given initial token distribution. In the following, we discuss the semantic match rules for comparing Petri net models according to the final marking equivalence criterion depicted in Listing 6.2 in detail.

In the match rule MatchTrace, first the final markings of the compared Petri net models are obtained from the trace models. In particular, the links, which exist in the end of the execution between the places of the Petri nets and tokens created during their execution for the association named *"holdingPlace"* are retrieved by invoking the operation getFinalHoldingPlaceLinks(). These links represent the information which tokens resided at which places in the end of the execution of the respective Petri net and, hence, the final marking of the Petri net. For retrieving them, the operation getFinalHoldingPlaceLinks() first selects all value instances captured by the trace models, which have not been destroyed during the execution and which capture the state of links, second selects those value instances that capture the state of links being instances of the association *"holdingPlace"*, third collects the respective last value snapshots of the selected value instances, and fourth collects the actual snapshot values, i.e., links.

Figure 6.3 illustrates this selection of links between places and tokens by depicting excerpts of the trace models $T_{M_1,7}$ and $T_{M_2,7}$ captured for the execution of the two example Petri net models for the inputs $I_{M_1,7}$ and $I_{M_2,7}$ comprising one token residing at Place *p2* and one token residing at Place *p3* (cf. Figure 6.2). These trace model excerpts show the value instances captured for the links we are interested in, namely the links representing the existence of tokens at places during the model execution. The respective first two value instances are captured for the initial tokens residing at *p2* and *p3* represented by the links `l1` and `l2` in the trace model $T_{M_1,7}$ and `l4` and `l5` in the trace model $T_{M_2,7}$. As for both Petri nets this initial marking leads to the firing of transitions, both value instances are destroyed during the execution, which is indicated by the existing destroyer references. In the first Petri net M_1, the transition *t2* is fired leading to a token residing at place *p4* constituting the final marking. This is represented by the third shown value instance for the link `l3`. Hence, the operation getFinalHoldingPlaceLinks() selects this link `l3`. In the second Petri net M_2, both transitions *t2* and *t3* are enabled due to the initial marking. The firing of *t2* leads to the destruction of the first value instance for the link `l4` representing that a token resides at *p2* and to the creation of the third value instance for the link `l6` representing that a token resides at *p3*. At this point in time of the execution, two tokens reside at place *p3*. Hence, the transition *t3* is fired two times leading to the destruction of these two tokens and the links `l5` and `l6` and to the creation of two tokens residing at place *p4* represented by the links `l7` and `l8`. As these two tokens represent the final marking of the Petri net M_2, the links `l7` and `l8` are selected by the operation getFinalHoldingPlaceLinks().

In the match rule MatchTrace, the retrieved links finalHoldingPlaceLinksLeft and finalHoldingPlaceLinksRight representing the final markings of the executed Petri nets are matched with each other. If for each link in finalHoldingPlaceLinksLeft a corresponding link exists in finalHoldingPlaceLinksRight and vice versa, both Petri nets reach the same final marking and are, hence, final marking equivalent. Thus, in this case, the value true is returned by the match rule.

The matching of links between places and tokens is performed by applying the rule MatchLink. Therefore, the places being linked by the two compared links are determined using the operation getLinkedObject(). They denote the places a token contained by the final markings of the Petri nets resided at. If the determined places match, both compared links represent the existence of a token at the same place and the rule returns true indicating that the links match.

The places being linked are matched by the rule MatchPlace. For understanding this rule we have to bear in mind that for the execution of Petri net models, they are translated into fUML extensional values by the generic model execution environment. Furthermore, the Petri net models have already been syntactically compared in the syntactic matching step leading to the establishment of correspondences between places and transitions contained by the compared Petri nets. Hence, first the original places placeLeft and placeRight contained by the Petri net models are obtained from the conversion result held by the generic model execution environment by calling the operation getInputObject(). Thereafter, it is checked whether a correspondence between these places has been established during the syntactic matching. If this is the case, the match rule returns true indicating that the two compared places being linked with a token match.

As shown in Figure 6.3, the semantic match rule MatchPlace establishes a match between the objects representing place *p4* of the two compared Petri net models M_1 and M_2. Furthermore, the semantic match rule MatchLink establishes a match between the links `l3` and `l7` representing that a token resides at place *p4* at the end of the execution of the two compared Petri net models. However, no match was established for the link `l8` representing that a second token resides at place *p4* at the end of the execution of the Petri net model M_2. Hence, the semantic match rule MatchTrace returns *false*, indicating that the two compared trace models $T_{M_1,7}$ and $T_{M_2,7}$ do not match. Thus, the Petri net models M_1 and M_2 are not final marking equivalent and the trace models $T_{M_1,7}$ and $T_{M_2,7}$, consequently, constitute diff witnesses.

In this section, we have presented the instantiation of our generic semantic model differencing framework for fUML-based behavioral semantics specifications and illustrated how semantic model differencing operators can be realized using this instantiation.

For fUML-based behavioral semantics specifications, semantic model differencing can be realized by utilizing the extended fUML execution environment for executing the models to be compared for given inputs and syntactic model differencing techniques for comparing the trace models captured for the performed model executions according to a semantic equivalence criterion suitable to the respective modeling language and its usage scenario. The only artifact that has to be implemented for realizing a specific semantic model differencing operator is the comparison algorithm for trace models implementing the semantic equivalence criterion to be applied in the semantic model differencing. We have shown how such comparison algorithms can be defined using declarative match rules. For applying a semantic differencing operator to two models, the models are executed for the same set of inputs leading to a set of trace models. If the trace models captured for the execution of the two compared models for the same input match according to the semantic match rules, the models are semantically equivalent with respect to the applied semantic equivalence criterion. Otherwise, if the trace models captured for at least one input do not match, the models are not semantically equivalent and the trace models that do not match constitute diff witnesses.

```
rule MatchTrace
   match left : Left!Trace with right : Right!Trace {
      compare {
         var finalHoldingPlaceLinksLeft : Set = left.getFinalHoldingPlaceLinks();
         var finalHoldingPlaceLinksRight : Set = right.getFinalHoldingPlaceLinks();
         return finalHoldingPlaceLinksLeft.matches(finalHoldingPlaceLinksRight) and
                finalHoldingPlaceLinksRight.matches(finalHoldingPlaceLinksLeft);
      }
}

operation Trace getFinalHoldingPlaceLinks() : Set {
   return self.valueInstances
      .select(vi | vi.destroyer = null and vi.runtimeValue.isTypeOf(Link))
         // select value instances of all links existing at termination
      .select(vi | vi.runtimeValue.type.name = "holdingPlace")
         // select value instances of links being instances of association "holdingPlace"
      .collect(vi | vi.snapshots.get(vi.snapshots.size() - 1))
         // collect last snapshots of value instances
      .collect(s | s.value);
         // collect values of snapshots (Link instances)
}

@lazy
rule MatchLink
   match left : Left!Link with right : Right!Link {
      compare {
         var placeLeft : Object = left.getLinkedObject("holdingPlace");
         var placeRight : Object = right.getLinkedObject("holdingPlace");
         return placeLeft.matches(placeRight);
      }
}

operation Link getLinkedObject(endName : String) : Object {
   var end : Property = self.type.memberEnd.select(me | me.name = endName);
   var objectReference : Reference = self.getFeatureValue(end).values.get(0);
   return objectReference.referent;
}

@lazy
rule MatchPlace
   match left : Left!Object with right : Right!Object {
      guard : left.isPlaceObject() and right.isPlaceObject()
      compare {
         var placeLeft : Place = conversionResult.getInputObject(left);
         var placeRight : Place = conversionResult.getInputObject(right);
         return placeLeft.matches(placeRight); // correspondences established in syntactic matching
      }
}

operation Object isPlaceObject() : Boolean {
   return self.types.select(t | t.name = "PlaceConfiguration").size() <> 0;
}
```

Listing 6.2: Semantic model differencing example: Semantic match rules based on fUML trace model for Petri nets (final marking equivalence)

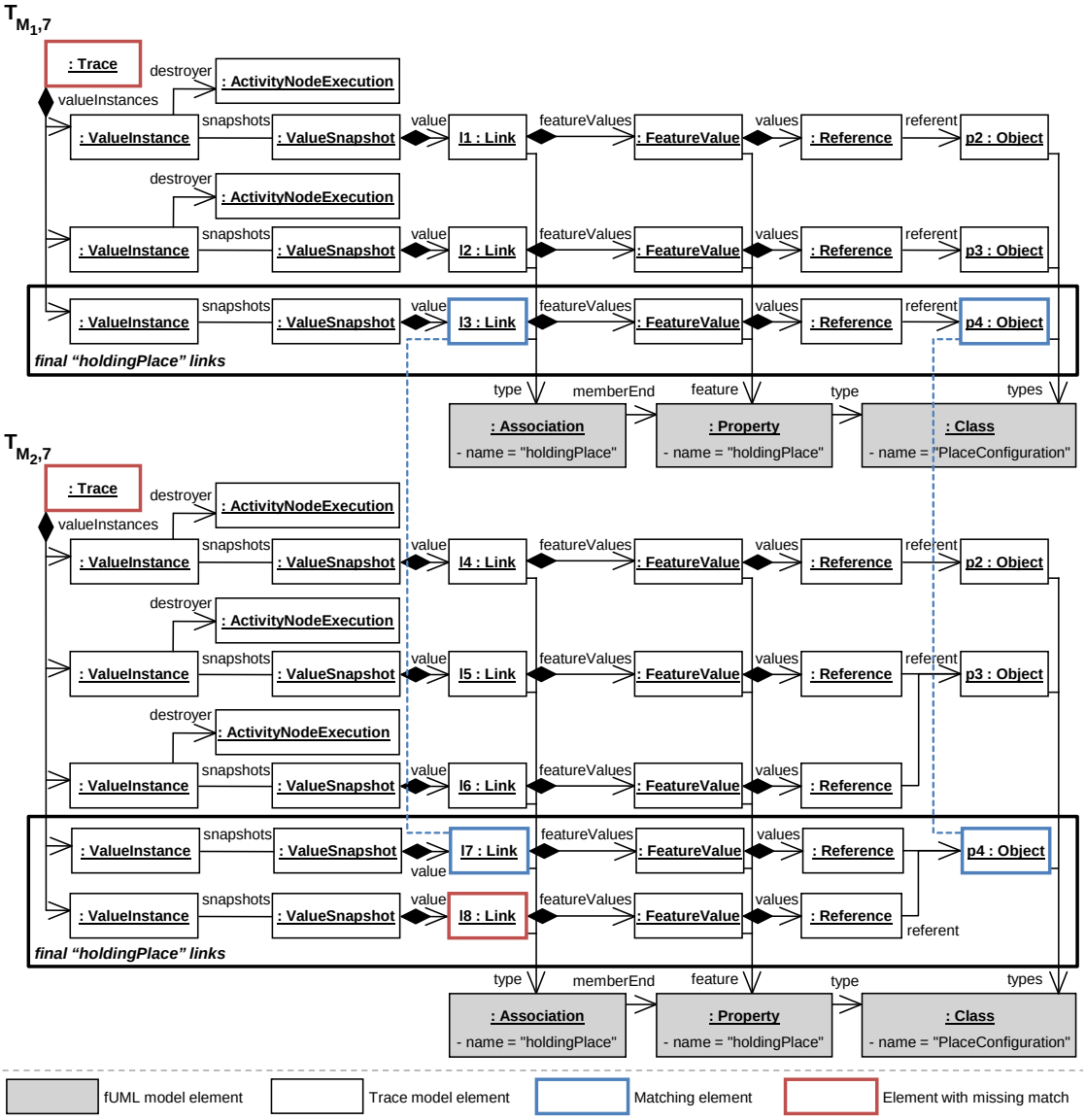


Figure 6.3: Semantic model differencing example: Excerpts of fUML trace models

6.4 Semantic Differencing for Operationally Defined Semantics Specifications

In the previous section, we have introduced the instantiation of our semantic model differencing framework for fUML-based behavioral semantics specifications. Using this instantiation it is possible to apply semantic differencing to models conforming to a modeling language whose behavioral semantics is defined using fUML. Thereby, the semantic model differencing is realized as comparison of trace models obtained from the execution of two compared models. These trace models can be retrieved from the extended fUML execution environment and adhere to an execution trace format specific to fUML-based behavioral semantics specifications.

However, retrieving execution traces required for realizing semantic model differencing following our approach is not only possible for fUML-based behavioral semantics specifications but for operationally defined behavioral semantics specifications in general. Therefore, either the execution environment provided by the respective operational semantics approach can be utilized, or the behavioral semantics specification itself can be manually or automatically extended for capturing execution traces.

To achieve genericity of our semantic model differencing framework with respect to the concretely applied semantics specification approach, we defined a *generic execution trace format*. This generic execution trace format serves as interface to our semantic model differencing framework. In particular, the semantic model differencing is realized as comparison of execution traces conforming to this format. Hence, through the introduction of the generic execution trace format, our semantic model differencing framework does not depend on a specific semantics specification approach or execution environment, but only operates on execution traces conforming to this format.

In the following, we present the generic execution trace format, discuss how execution traces adhering to this format can be retrieved from behavioral semantics specifications defined with different operational semantics approaches, and describe how semantic model differencing is realized based on execution traces conforming to the generic format in more detail.

6.4.1 Generic Execution Trace Format

The generic execution trace format used for realizing semantic model differencing based on behavioral semantics specifications defined using any operational semantics approach is defined by the metamodel depicted in Figure 6.4. A trace (metaclass Trace) consists of a set of states (metaclass State), which capture the runtime state of a model at a specific point in time of its execution. Therefore, a state consists of snapshots of each model element's state (reference objects). In an operational semantics approach, the state of model elements is captured by instances of the runtime concepts defined in the semantics specification (cf. Section 2.2.2). If EMF is used as metamodeling environment for defining runtime concepts, these instances are instantiations of Ecore's metaclass EObject. Transitions between consecutive states of an executed model (metaclass Transition) are labeled with the event (metaclass Event) that caused the respective state change leading from one state to another state (references source and target).

Execution traces conforming to this generic format represent the runtime behavior of models

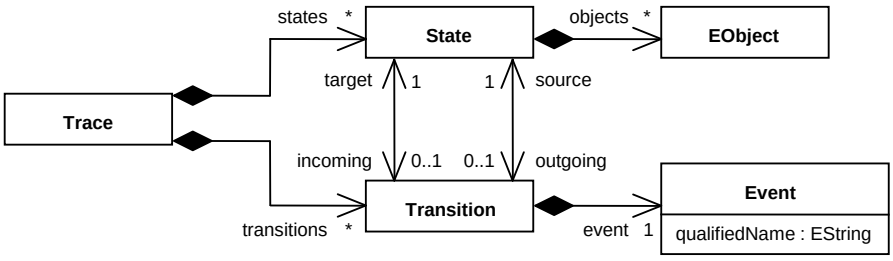


Figure 6.4: Generic execution trace format for semantic model differencing

in terms of sequences of the models' runtime states during their execution. Thus, they constitute traces through the state transition systems defined by the used modeling languages' operationally defined behavioral semantics specifications. By analyzing such execution traces, it is possible to determine each runtime state of an executed model, transitions between these runtime states, as well as events causing state transitions.

6.4.2 Capturing Execution Traces

For capturing execution traces adhering to the generic execution trace format introduced in the previous section, means for accessing the runtime state of an executing model are required. This means might either be provided by the execution environment of the respectively employed operational semantics approach or introduced into the behavioral semantics specification itself. The utilization of the execution environment for capturing execution traces clearly is to be preferred over the extension of the behavioral semantics specification. In the following, we discuss for three distinct operational semantics approaches, namely fUML, Kermeta, and DMM, how execution traces complying to the defined format can be captured.

Capturing Execution Traces with fUML

As the execution environment for fUML-based behavioral semantics specifications is built upon the extended fUML execution environment, it provides means for observing the runtime state of a model during its execution as well as means for dynamically analyzing the runtime states of a model after its execution finished. For the former, the event mechanism of the extended fUML execution environment can be utilized, for the latter, a trace model can be retrieved from the extended fUML execution environment. Both enable the construction of execution traces adhering to the defined generic execution trace format.

For our implementation, we chose the former approach, namely the utilization of the *event mechanism* provided by the extended fUML execution environment for capturing execution traces. The event mechanism issues an *extensional value event* whenever the runtime state of an executing model changes due to the execution of fUML actions, such as create object actions, destroy object actions, and add structural feature value actions. Thus, when receiving an extensional value event, our implementation creates a new state and adds it to the execution trace, copies the current runtime state of the model, and adds this copy to the newly created

state. Furthermore, a transition from the last state already contained by the execution trace to this newly created state is added to the execution trace and an event is created for this transition. Thereby, the event of the transition is labeled with the full qualified name of the action whose execution caused the current runtime state change, i.e., whose execution led to the issuing of the received extensional value event. This action can again be determined by utilizing the event mechanism. In particular, the very last event issued by the event mechanism before the issuing of an extensional value event is an *action entry event* indicating the start of the execution of the action causing the respective change of the model's runtime state.

Alternatively, execution traces adhering to the defined generic format could be constructed from the *trace models* provided by the extended fUML execution environment for performed model executions. In particular, from the *value snapshots* captured by a trace model, all runtime states of the executed models as well as the information which action execution led to the respective runtime state can be retrieved. Thus, based on the value snapshots and action executions captured in a trace model, states, transitions, and events can be accordingly derived and captured in an execution trace.

Capturing Execution Traces with Kermeta

To capture execution traces for model executions performed based on behavioral semantics specifications defined with Kermeta, two alternative approaches may be applied.

The first alternative is to *extend the behavioral semantics specifications* of modeling languages, such that execution traces are captured during the execution of conforming models. Therefore, Kermeta's aspect weaving capabilities as well as action language may be used. In particular, operations have to be introduced into the behavioral semantics specification of a modeling language, which are responsible for retrieving the current runtime state of an executing model and adding a new state to the execution trace. Furthermore, the operations defined as part of the behavioral semantics specification themselves have to be extended, such that after each statement, which changes the runtime state of an executing model, the operations responsible for updating the execution trace are called. These required extensions of behavioral semantics specifications may either be performed manually or achieved in an automated way. Automating the required extensions is possible due to fact that Kermeta programs are models themselves conforming to an Ecore-based metamodel. Thus, by statically analyzing a Kermeta-based behavioral semantics specification, statements which change the runtime state of an executing model may be determined and additional statements may be introduced, which cause the update of an execution trace. Thereby, the update of an execution trace may be implemented generically in its own class, which is then reused for any behavioral semantics specification defined with Kermeta.

The second alternative for capturing execution traces is to leverage Kermeta's integration with EMF. In particular, by utilizing *EMF's notification and adapter framework*, it is possible to receive notifications about each change applied to a model. Thus, by adding an adapter to the model being executed, each update of the model's runtime state may be observed. This enables the construction of execution traces adhering to our generic format.

Capturing Execution Traces with DMM

When DMM is used for defining the behavioral semantics of modeling languages, execution traces adhering to our generic execution trace format may be constructed by utilizing DMM's execution environment, namely the GROOVE tool set. Therefore, we may apply two alternative approaches discussed in the following.

In the first approach, we may utilize the GROOVE tool set for applying the graph transformation rules defining the behavioral semantics of the respective modeling language in a stepwise manner leading to a *stepwise execution* of a model. When doing so, the host graph representing the current runtime state of the executing model is stepwise updated. Thus, after each rule application, the current state of the host graph may be retrieved, translated back to the runtime metamodel defined as part of the semantics specification, and added to a newly created state in the execution trace. Thereby, transitions between states may be labeled with the name of the graph transformation rule applied in the last execution step.

Alternatively, an execution trace complying to our generic execution trace format may be constructed from the *labeled transition system* generated by the GROOVE tool set for a performed model execution. This labeled transition system represents each possible runtime state of an executed model in terms of host graphs as well as transitions between these states, which are labeled with the name of the applied graph transformation rule leading to the respective state transition. Thus, for each trace through the generated labeled transition system, one execution trace adhering to our generic format may be derived. In particular, for each state in the labeled transition system, one state is added to the execution trace. Thereby, the host graph of the respective state in the labeled transition system is translated back to the runtime metamodel and the resulting representation of the model's runtime state is added to the newly created state in the execution trace. Furthermore, for the new state a transition as well as an event for this transition are added to the execution trace. Thereby, the event is labeled with the name of the graph transformation rule leading to the respective state transition, which can be easily retrieved from the labeled transition system.

6.4.3 Semantic Model Differencing Based on Generic Execution Traces

In the previous section, we have shown that it is in general possible to retrieve execution traces adhering to our generic execution trace format from operationally defined behavioral semantics specifications. As will be described in the following, execution traces complying to this format enable the realization of semantic model differencing in a generic manner. By relying only on this generic execution trace format, our semantic model differencing framework becomes generally applicable irrespective of the operational semantics approach used for defining the behavioral semantics of a modeling language. For instantiating the framework for a specific operational semantics approach, it is only required that this semantics specification approach provides means for constructing execution traces complying to the generic execution trace format. In the following, we recap the three steps involved in the semantic model differencing and detail the semantic matching step performed on execution traces adhering to our generic format.

1. Syntactic matching. In the syntactic matching step, two models M_1 and M_2 are syntactically compared by utilizing existing generic model differencing techniques. As this step is only concerned with the abstract syntax of the compared models, it remains unaffected by the introduction of the generic execution trace format. Our implementation of the syntactic matching step is based on ECL.

2. Model execution. In the model execution step, the two models M_1 and M_2 are executed for inputs I_{M_1} and I_{M_2}. Thereby, the execution is performed based on the used modeling language's behavioral semantics specification by utilizing the model execution environment provided by the respectively used semantics specification approach, such as the execution environment of fUML in the case of fUML-based behavioral semantics specification, the execution environment of Java in the case of Kermeta-based behavioral semantics specifications, and the execution environment of the GROOVE tool set in the case of DMM-based behavioral semantics specifications. During the model execution, execution traces T_{M_1} and T_{M_2} adhering to our generic execution trace format are captured, which are handed over to the semantic matching after the completion of the execution.

3. Semantic matching. In the semantic matching step, the execution traces T_{M_1} and T_{M_2} obtained from the model execution step are compared by the application of semantic match rules. If multiple inputs are considered in the semantic differencing, the execution traces are pairwise compared. This means that the execution traces obtained from executing the two compared models on the same input are compared with each other by applying the semantic match rules. The semantic match rules compare the elements contained by the execution traces and establish correspondences between matching execution trace elements. As the execution traces adhere to our generic execution trace format, the runtime states of the executed models as well as the state transitions captured by the execution traces are compared by the semantic match rules. If a correspondence between the two execution traces can be established, the two compared models behave equivalently for the respective input and are, hence, semantically equivalent for this input. Thus, if correspondences between all pairs of execution traces obtained for all considered inputs can be established, the compared models are semantically equivalent. Otherwise, execution traces for which no correspondences could be established constitute diff witnesses.

Compared to the semantic matching of trace models for fUML-based behavioral semantics specifications discussed in Section 6.3, only the match rules themselves are affected by the introduction of the generic execution trace format. In particular, the semantic matching is now performed on the runtime state of an executing model, which does not have to be extracted from the fUML-specific trace model but is directly captured by the execution traces. As will be illustrated in the following example, this eases the implementation of semantic match rules. Our implementation relies on ECL for defining and applying semantic match rules.

Example. In the following, we discuss based on our Petri net example the definition of semantic match rules for execution traces complying to our generic execution trace format. In this example, we consider two different semantic equivalence criteria for Petri nets, namely *final marking equivalence*, which we already considered in Section 6.3, and *marking equivalence*,

which we adopted from the literature [46]. Two Petri net models with the same set of places are final marking equivalent if they reach for the same initial marking the same final marking. They are marking equivalent, if they reach for the same initial marking the same set of markings.

Listing 6.3 shows the semantic match rules expressed in ECL for determining whether two Petri net models are *final marking equivalent*. These semantic match rules are equivalent to those defined in Listing 6.2 but instead of matching fUML-specific trace models, they match execution traces adhering to our generic execution trace format for performing the semantic model differencing. The rule MatchTrace matches execution traces captured for the execution of two compared Petri net models for a given initial token distribution. Therefore, the final runtime states of both Petri net models are obtained from the execution traces using the operation getMarkingStates(). If these final runtime states markingStatesLeft and markingStatesRight match, the Petri net models are final marking equivalent for the given initial token distribution, and, consequently, the rule MatchTrace returns true. The matching of the retrieved states is done by the rule MatchState. This rule retrieves the runtime states of all places represented by PlaceConfiguration instances captured by the two matched states, by calling the operation getPlaceConfigurations(). If they match, the rule MatchState returns true and a correspondence between the two compared states is established. The runtime state of the PlaceConfiguration instances are matched by the rule MatchPlaceConfiguration. It defines that two PlaceConfiguration instances match, if they hold the same amount of tokens. Thus, in summary, the rule MatchTrace returns true, if in the last runtime state of the Petri net models all places hold the same amount of tokens, i.e., if the two compared Petri net models have the same markings in the end of the execution.

For realizing the *marking equivalence criterion*, only the operation getMarkingStates() has to be adapted as shown in Listing 6.4. It retrieves the runtime state of the executed Petri net model after the activity NetConfiguration::initializeMarking has been executed and after each execution of the activity TransitionConfiguration::fire. Therefore, the utility operation getStatesAfterEvent() provided by our implementation of the generic execution trace format is used. It retrieves the states caused by an event corresponding to the provided qualified name. Thus, the operation getMarkingStates() returns the runtime state of a Petri net model after its marking has been initialized, as well as after each transition firing, i.e., each state after reaching a new marking. These sets of states markingStatesLeft and markingStatesRight are matched by the rule MatchTrace (cf. Listing 6.3). If each state in markingStatesLeft has a corresponding state in markingStatesRight and vice versa, that is, if each marking reachable in M_1 is also reachable in M_2 and vice versa, a correspondence is established between the two execution traces and, hence, the considered Petri net models are marking equivalent.

Figure 6.5 depicts excerpts of the execution traces $T_{M_1,7}$ and $T_{M_2,7}$ obtained from the execution of the Petri net models M_1 and M_2 on the initial token distribution $I_{M_1,7}$ and $I_{M_2,7}$ (cf. Figure 6.2). The respective last states `s8` and `s15` of the execution traces represent the final runtime states of the executed models. These states are compared in case the final marking equivalence criterion is applied. Because in `s8` one token resides at place *p4* and in `s15` two tokens reside at place *p4*, these two states do not match and a correspondence is neither established for these states nor for the execution traces. Thus, the compared execution traces constitute diff witnesses.

```
rule MatchTrace
   match left : Left!Trace with right : Right!Trace {
      compare {
         var markingStatesLeft : Set = left.getMarkingStates();
         var markingStatesRight : Set = right.getMarkingStates();
         return markingStatesLeft.matches(markingStatesRight) and
                markingStatesRight.matches(markingStatesLeft);
      }
}

operation Trace getMarkingStates() : Set {
   return self.states.at(self.states.size() - 1).asSet();
}

@lazy
rule MatchState
   match left : Left!State with right : Right!State {
      compare {
         var placeConfsLeft : Set = left.getPlaceConfigurations();
         var placeConfsRight : Set = right.getPlaceConfigurations();
         return placeConfsLeft.matches(placeConfsRight);
      }
}

operation State getPlaceConfigurations() : Set {
   var placeConfs : Set = new Set();
   for (object : Any in self.objects)
         if (object.isKindOf(PlaceConfiguration))
            placeConfs.add(object);
   return placeConfs;
}

@lazy
rule MatchPlaceConfiguration
   match left : Left!PlaceConfiguration with right : Right!PlaceConfiguration
   extends MatchPlace {
      compare : left.heldTokens.size() = right.heldTokens.size()
}
```

Listing 6.3: Semantic model differencing example: Semantic match rules based on generic execution trace format for Petri nets (final marking equivalence)

```
operation Trace getMarkingStates() : Set {
   var markingStates : Set = new Set();
   markingStates.addAll(self.getStatesAfterEvent("NetConfiguration::initializeMarking"));
   markingStates.addAll(self.getStatesAfterEvent("TransitionConfiguration::fire"));
   return markingStates;
}
```

Listing 6.4: Semantic model differencing example: Adaptation of semantic match rules based on generic execution trace format for Petri nets (marking equivalence)

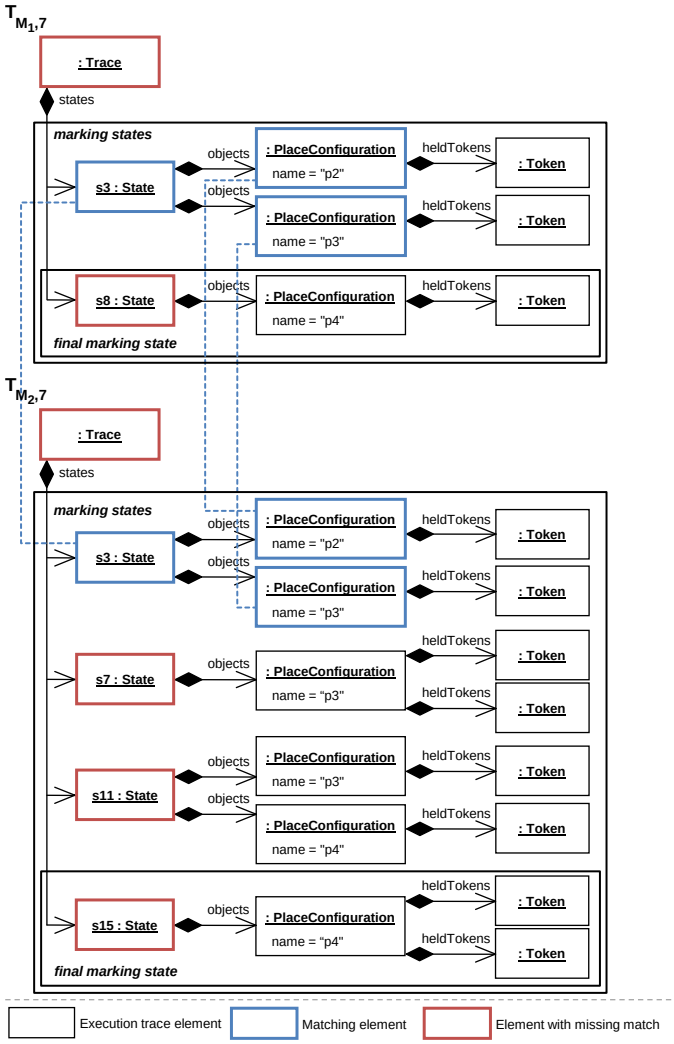


Figure 6.5: Semantic model differencing example: Excerpts of generic execution traces

When the marking equivalence criterion is applied, not only the final runtime states of the compared Petri net models but all runtime states after reaching a new marking are matched. Figure 6.5 depicts these runtime states to be matched for the example models. The states `s3` and `s8` of $T_{M_1,7}$ are matched by the semantic match rules with the states `s3`, `s7`, `s11`, and `s15` of $T_{M_2,7}$. Because only the states `s3` of $T_{M_1,7}$ and `s3` of $T_{M_2,7}$ match, no correspondence between the execution traces can be established, and, hence, the Petri net models are not marking equivalent, which is witnessed by these execution traces.

6.5 Input Generation for fUML-based Semantics Specifications

Semantic model differencing operators realized with our generic framework rely on *concrete execution traces* obtained by executing the models to be compared for *concrete inputs*. In the semantic differencing, the execution traces obtained by executing the models for the same inputs are compared by applying semantic match rules. If for all compared execution traces a correspondence can be established, the models are semantically equivalent. Otherwise, non-corresponding execution traces constitute diff witnesses being manifestations of the semantic differences among the models.

In the previous sections, we have illustrated our semantic model differencing framework based on the example of the Petri net language. A Petri net takes as input an initial token distribution that can contain arbitrary many tokens residing at places of the Petri net. The execution trace obtained from executing a Petri net on a specific initial token distribution captures information about each marking reached during the execution due to the firing of transitions. In the semantic differencing, two Petri nets are compared based on this information captured in the obtained execution traces. However, as the number of possible initial token distributions is infinite, it is not possible to execute the compared Petri nets for all possible inputs, obtain all resulting execution traces, and use these execution traces for semantically differencing the models.

Enumerating all possible inputs and performing the semantic differencing for all execution traces resulting from these inputs is not feasible for several scenarios, as the number of possible inputs may quickly become large or even infinite. However, we may restrict the inputs considered for the semantic model differencing to those inputs that cause *distinct execution traces* as only distinct execution traces may lead to the identification of semantic differences among models. Having obtained such inputs, the models to be compared can be executed for these inputs and the semantic match rules can be applied to the captured execution traces for semantically differencing the models.

For *automatically* generating such relevant inputs and, hence, automating the semantic model differencing, the behavioral semantics specification of the considered modeling languages may be analyzed. This section is concerned with generating inputs from fUML-based behavioral semantics specifications. In particular, we propose to apply an adaptation of symbolic execution [22] to fUML.

6.5.1 Symbolic Execution for fUML

The basic idea behind symbolic execution, as introduced by Clarke [25], is to execute a program—in our case an fUML model—with *symbolic values* in place of concrete values. For each conditional statement that is evaluated over symbolic values along an execution path, a *path condition* is recorded in terms of a quantifier-free first-order formula. Thereby, for each symbolic value, a *symbolic state* is maintained during the symbolic execution, which maps symbolic values to symbolic expressions. After executing a path symbolically, we obtain a sequence of path conditions, which can be conjuncted and solved by a constraint solver to obtain concrete inputs. An execution with these inputs will consequently exercise the path that has been recorded symbolically. If a conjunction of path conditions is unsatisfiable, the execution path can never occur. Using *backtracking* and *negations of path conditions*, we may obtain *all feasible paths*. These

paths may be represented as an *execution tree*, which is a binary tree consisting of nodes denoting path conditions and edges denoting Boolean values.

More recently, several extensions and flavors of traditional symbolic execution have been proposed as surveyed by Cadar and Sen [22]. For symbolically executing fUML-based behavioral semantics specifications we propose to apply a combination of concolic execution [141] and generalized symbolic execution [76]. *Concolic execution* significantly decreases the number of path conditions by distinguishing between *concrete values and symbolic values*. The program is essentially executed as normal and only statements that depend on symbolic values instead of concrete values are handled differently. In our fUML-based semantics specification approach, we execute the semantics specification for a concrete model and a concrete input. Thus, applying concolic execution, we may consider only the input as symbolic and statements that interact with the executed model itself are executed as normal. One of the key ideas behind *generalized symbolic execution*, which we also propose to apply, is to use *lazy initialization* of symbolic values. Thus, we execute the model as normal and initialize empty objects for symbolic values only when the execution accesses the objects for the first time. Similarly, attribute values of objects are only initialized on their first access during the execution with dedicated values to induce a certain path during the execution.

In the following, we discuss based on our Petri net example how fUML-based behavioral semantics specifications can be symbolically executed for concrete models in order to obtain concrete inputs for these models that can be used in the semantic model differencing.

Example. To derive concrete inputs for a Petri net model that cause all distinct execution traces, we symbolically execute the fUML-based behavioral semantics specification of the Petri net language (cf. Figure 5.6 and Figure 5.7) for this model. The input of the execution, that is the initial token distribution provided through the input parameter initialTokens of type EList<Token> of the activity main, is defined as a symbolic value.

Figure 6.6 shows an excerpt of the resulting execution tree for the Petri net model M_1 (cf. Figure 6.2). Please note that we bound the symbolic execution to at most one initial token per place in this example. We depict path conditions as diamonds and the symbolic states of symbolic values in boxes. Symbolic values are prefixed with a $ symbol.

The uppermost box shows the symbolic states after executing the operation initializeMarking(). This operation assigns the initial tokens to places of the Petri net by accordingly initializing the places' values for the heldTokens reference. As this is done based on the symbolic input initialTokens, also the values assigned for this reference are handled symbolically. The initial symbolic values for this reference are mapped to the symbolic expression `$initialTokens->select(t | t.holdingPlace = pX)`. In Figure 6.6, this expression is abbreviated with `$pXTokens`.

After the marking is initialized, the operation run() is called. This operation iterates over the transitions of the Petri net and checks whether they are enabled. Therefore, in the first iteration, isEnabled() is called on transition *t1*, which in turn iterates over all of its incoming places (*p1* in our example) and checks whether there is an incoming place without tokens. Therefore, the symbolic value `p1.heldTokens` is accessed and the condition `p1.heldTokens.size() == 0` is evaluated. We do not interfere with the concrete execution, except for the access of

the symbolic value and the evaluation of the condition in order to record the path condition, update the execution tree, and solve the condition to compute concrete values for the involved symbolic values inducing the `true` branch and the `false` branch. After that, we continue with the concrete execution for one of the branches. Depending on which branch is taken (i.e., *t1* is enabled or not), *t1* is added to the output expansion node enabled of the expansion region named *"select enabled transitions"* in the activity run. In the symbolic execution of activities, we handle expansion nodes as list variables. As the addition of *t1* to the expansion node depends on symbolic values, we also consider the list variable, denoted with `$enabled`, as symbolic.

In the next iteration of run, the same procedure is applied to transition *t2* and its input places *p2* and *p3*, and the execution tree is updated accordingly.

Next, the execution checks whether the list of enabled transitions contains at least one element with the condition `$enabled.size() > 0`. As this condition accesses `$enabled`, which is considered as symbolic value, we record it in the execution tree and try to produce values for the `true` and `false` branch. For the first three paths, the constraint solver cannot find a solution for the `false` branch, denoted with $\perp$, because in these paths `$enabled` will always contain at least one transition according to the path conditions and symbolic states. Thus, in three of the six branches contained by these three paths, fire() is called for the first transition in the list `$enabled` causing changes of the heldTokens reference values of its incoming and outgoing places. As the heldTokens reference values are considered as symbolic, we update their symbolic states. Finally, the execution proceeds with iterating through transitions again and firing them, if they are enabled. As we bound the symbolic execution to at most one initial token per place, all branches either terminate eventually or lead to an unsatisfiable state (e.g., violating the bound constraint). In the fourth path not discussed so far, the condition `$enabled.size() > 0` can only be satisfied for the `false` branch, which leads to the termination of the execution.

The final execution tree contains four satisfiable execution paths. The path condition of these paths represent symbolically all relevant initial token distributions for this Petri net inducing all distinct execution traces. Using a constraint solver, we can generate Token objects with corresponding links to the places in the Petri net, such that the we obtain the following initial markings: $\{p1 = 1\}$, $\{p2 = 1, p3 = 1\}$, $\{p1 = 1, p2 = 3, p3 = 1\}$, and $\{\}$ (no tokens at all). When repeating the symbolic execution for the Petri net M_2 (cf. Figure 6.2), we obtain four additional inputs: $\{p2 = 1\}$, $\{p3 = 1\}$, $\{p1 = 1, p2 = 1\}$, and $\{p1 = 1, p3 = 1\}$. With this total of eight inputs, we invoke the semantic differencing in which the Petri net models are executed for these inputs and compared by applying the semantic match rules to the captured execution traces (cf. Section 6.3 and Section 6.4). As shown in Figure 6.2, six of the generate inputs lead to the identification of diff witnesses when applying the final marking equivalence criterion.

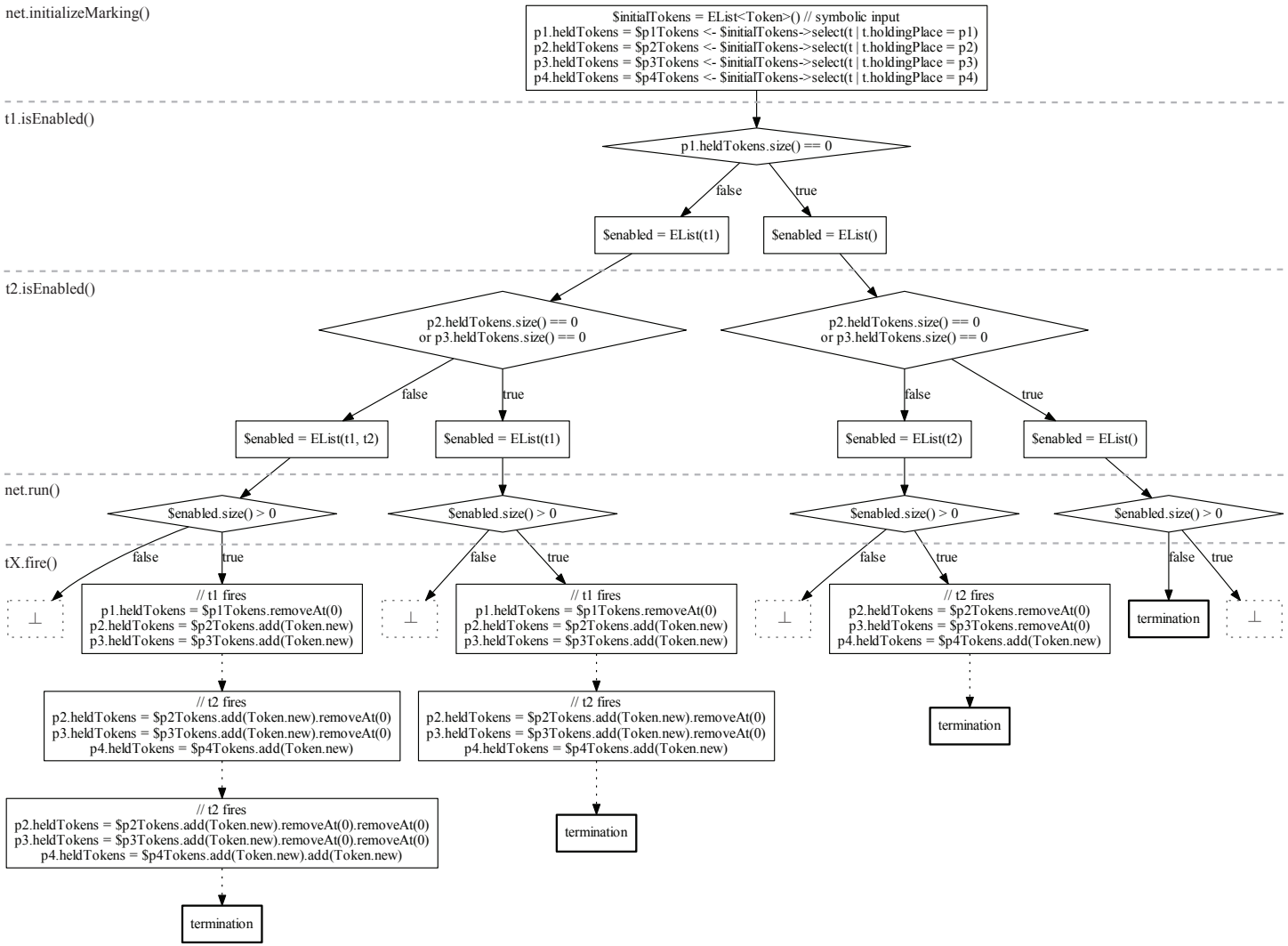


Figure 6.6: Symbolic execution example: Execution tree

In this section, we have discussed how symbolic execution can be applied to fUML-based behavioral semantics specifications to obtain execution trees allowing the generation of input values leading to the execution of all possible paths through the semantics specification for a given model. The input values generated for two models to be compared are provided as input to the semantic model differencing framework, which executes the models to be compared for all generated inputs and compares the obtained execution traces by applying semantic match rules. However, please note that the execution trees obtained from the symbolic execution already capture all information required for performing the semantic model differencing, as they capture all possible paths through the semantics specification as well as all possible states of the models to be compared. Thus, the execution of the models to be compared on the concrete inputs generated from the obtained execution trees may be omitted and instead, the semantic model differencing may be performed based on the execution trees. This possibility is related to the technique *differential symbolic execution* proposed by Person *et al.* [126] and constitutes a future research direction that has to be further investigated.

6.5.2 Implementation of Symbolic Execution for fUML

Realizing the proposed approach for generating inputs from fUML-based behavioral semantics specifications requires the implementation of an environment enabling the *symbolic execution* of fUML-based behavioral semantics specification as well as the implementation of a *constraint solver* for generating inputs satisfying the path conditions computed by the symbolic execution. Thereby, the symbolic execution of an fUML-based behavioral semantics specification is performed on a concrete model yielding path conditions that represent all possible execution paths through the semantics specification for this model. The constraint solver takes as input the obtained path conditions being quantifier-free first-order formulas, solves them by computing satisfying Boolean variable assignments, and generates inputs from these Boolean variable assignments. In the following, we discuss three alternative approaches for implementing symbolic execution for fUML-based behavioral semantics specifications as well as a concrete implementation for generating inputs from path conditions through constraint solving.

Symbolic Execution

We investigated three alternative approaches for implementing the introduced symbolic execution for fUML-based behavioral semantics specifications. In the first approach, a *specialized symbolic execution environment for fUML* is implemented that allows the symbolic execution of fUML-based behavioral semantics specifications. In the second approach, an *existing symbolic execution environment* that allows the symbolic execution of Java programs is utilized for symbolically executing the Java-based fUML virtual machine whose execution underlies the execution of fUML-based behavioral semantics specifications. In the third approach, a code generator is implemented that allows generating *symbolically executable programs*, i.e., programs that can be symbolically executed by an existing symbolic execution environment, from fUML-based behavioral semantics specifications. In the following, we describe each of these alternatives in more detail and discuss their advantages and disadvantages.

Symbolic execution environment for fUML. A symbolic execution environment needs to (i) distinguish concrete and symbolic values, (ii) maintain the symbolic state of symbolic values by evaluating expressions over symbolic values, (iii) record path conditions, and (iv) backtrack executions to cover all possible execution paths. For fUML such a symbolic execution environment may be implemented based on fUML's execution environment as well as the introduced event mechanism and command interface as described in the following.

Before the symbolic execution of an fUML model is started, the data structure for maintaining the symbolic state of symbolic values as well as the data structure for maintaining the execution tree capturing the explored execution paths are initialized. Thereafter, the execution of the model is started in the *stepwise* execution mode using the command interface.

After the completion of each execution step, which is communicated via a suspend event issued by the event mechanism, it is determined whether the next execution step constitutes a *conditional step*. A conditional step is a step depending on a symbolic value, such as the execution of a decision node that receives a symbolic value as decision input value. In case a conditional step is identified, a path condition is added to the current execution path in the execution tree. For resuming the symbolic execution, one branch for the added path condition is chosen according to some exploration strategy, i.e., either the true branch or the false branch is chosen for the added condition. Using a constraint solver, a concrete value for the symbolic value is computed from the conjunction of all path conditions leading to the selected branch. This symbolic value is then inserted into the execution model and the execution is resumed.

Besides checking for conditional steps, also *writing actions* performed in the last execution step are further investigated. In particular, it is investigated whether a symbolic value was involved in the update performed by the writing action. If the writing action assigned a symbolic value to some variable, such as an attribute value, which is not yet declared as symbolic, it is declared as being symbolic from this time in the execution on and its state is accordingly initialized with a symbolic expression. If the writing action assigns a value to a symbolic value, the state of this symbolic value is accordingly updated.

If the symbolic execution reaches either an unsatisfiable path or the execution terminates, the execution is backtracked for further exploring heretofore unexplored paths. For the backtracking we may apply two distinct approaches. In the first approach, concrete values are computed for symbolic values existing in the selected path by solving the respective path conditions. Then the execution is started from the beginning and stepwise resumed using the command interface until the last considered conditional step is reached. While this approach can be easily implemented based on the event mechanism and command interface, it is in general expensive. The second much more efficient way to achieve backtracking is to restore the execution state at reaching the last conditional step of the selected execution path to be explored. However, this requires the implementation of an efficient mechanism for storing the execution state of an fUML model.

The advantage of this approach for realizing symbolic execution of fUML models is that the symbolic execution is by construction completely conform to the fUML standard as the concrete execution is performed by the standardized execution environment of fUML. The symbolic execution only inserts values, namely concrete values for symbolic values, into the execution model on which the fUML virtual machine operates. However, the major drawback of this approach is the required high implementation effort originating not only from the implementation of the

basic symbolic execution functionalities for fUML, such as backtracking and constraint solving techniques, but also from the required optimizations of both the fUML virtual machine as well as the symbolic execution for fUML required for coping with scalability issues and state explosion problems inherent to symbolic execution. Due to these issues, we have for now abandoned the implementation of an own symbolic execution environment for fUML. However, a promising approach for mitigating these issues was recently proposed by Bucur *et al.* [21]. They present a tool called CHEF that enables the building of symbolic execution engines for interpreted languages directly from their interpreters. The goal of CHEF is on the one hand to reduce the implementation effort of building a symbolic execution engine for an interpreted language and on the other hand to enable the construction of symbolic execution engines that are more efficient than directly executing the language's interpreter symbolically (which is the implementation alternative discussed next).

Symbolic execution of fUML virtual machine. As a second alternative for realizing symbolic execution of fUML-based behavioral semantics specification, we may apply symbolic execution to the fUML virtual machine itself. For understanding how this may be achieved, let us recap how fUML-based behavioral semantics specifications are executed on a model. To perform the execution, the semantics specification—which is defined using an executable metamodeling language integrating fUML, such as xMOF—is converted into an fUML model and the model to be executed as well as the input to the model are converted into fUML extensional values being instances of classes contained by the fUML model. The fUML extensional values obtained from the conversion of the model and the input are added to the locus of the fUML virtual machine. Thereafter, the actual model execution is started by calling the operation execute() of the single Executor instance of the fUML virtual machine. The fUML extensional values representing the input to the model are provided as parameters values. For symbolically executing the fUML virtual machine with the aim of generating inputs, we may define all values involved in the execution as concrete values, except the fUML extensional values that represent the input and are provided as parameter values to the operation Executor.execute(). Thus, the fUML model representing the fUML-based behavioral semantics specification, the fUML extensional values representing the model, the fUML execution environment objects (e.g., the Executor instance), and all semantic visitor objects instantiated by the fUML virtual machine are treated as concrete values. Thus, every statement of the fUML virtual machine that operate on those values only are executed as normal. Only the parameter values representing the input to the model are treated as symbolic values requiring symbolic state maintenance and execution path recording.

For performing this symbolic execution, we may utilize an existing symbolic execution environment for Java that supports concolic execution, i.e., that supports distinguishing between concrete and symbolic values and executes all program statements operating exclusively on concrete values as normal and only treats statements operating on symbolic values as symbolic. Two examples of such symbolic execution environments are Java Pathfinder[1] [76, 161] developed at the NASA Ames Research Center and jCUTE[2] [141, 142] developed at the University of Illinois. We have performed several experiments with Java Pathfinder to symbolically execute

[1] `http://babelfish.arc.nasa.gov/trac/jpf`, accessed 03.09.2014

[2] `http://osl.cs.illinois.edu/software/jcute`, accessed 03.09.2014

fUML models by symbolically executing the fUML virtual machine. These experiments led us to the identification of several technical hurdles that have to be overcome in order to generate inputs through the symbolic execution of the fUML virtual machine. For example, Java Pathfinder provides only limited support for String values and no support for lists of variable size. Furthermore, the Java program to be symbolically executed has to be monolithic. As our current execution environment for fUML-based behavioral semantics specifications is realized as a set of Eclipse plugins that depend on other Eclipse plugins, adaptations of our implementation are required for enabling the utilization of Java Pathfinder. In particular, it requires the generation of Java code that constructs the fUML model and fUML extensional values representing the fUML-based behavioral semantics specification and the model, respectively, initializes the input and declares it as symbolic, and starts the execution of the fUML model by calling the fUML virtual machine.

The advantage of this approach is that existing symbolic execution environments may be reused, which are already highly optimized for symbolically executing programs written in a certain programming language. Symbolic execution environments exist also for programming languages other than Java [22], such as C, C++, and .NET languages. Thus, if an fUML virtual machine written in another language than Java is used, symbolic execution of fUML-based behavioral semantics specifications may still be performed. Also symbolic model checkers, such as SPIN and NuSMV, could be used to realize this approach. However, it requires the specification of fUML's behavioral semantics with the language supported by the respective model checker, such as PROMELA and SMV. The disadvantage of this approach is that instead of exploring all possible paths through the fUML-based behavioral semantics specification for a given model all possible paths through the fUML virtual machine for the fUML-based behavioral semantics specification are explored. Thus, applying this implementation alternative potentially leads to a much higher number of explored paths and generated path conditions, thus, resulting in the generation of a higher number of inputs to be considered in the semantic model differencing than in the first implementation alternative described beforehand.

Generation of a symbolically executable program. As a third alternative approach for implementing the input generation through symbolic execution, we may implement a code generator that takes as input an fUML-based behavioral semantics specification as well as a model and produces a program that can be symbolically executed by utilizing existing symbolic execution environments. In particular, the generated code comprises three parts, namely classes and operations that correspond to the fUML-based behavioral semantics specification, code that instantiates the generated classes such that the resulting objects correspond to the model for which inputs shall be generated, and code that declares the input to the model as symbolic and starts the model execution.

We have experimented with the generation of suitable Java code as well as the symbolic execution of this code with Java Pathfinder. In particular, we have manually implemented Java code corresponding to the fUML-based behavioral semantics specification of our example Petri net language as well as Java code that instantiates Java objects corresponding to our example Petri net models and that starts the execution of the model. The symbolic execution of the resulting program with Java Pathfinder leads to the generation of the expected path conditions.

The advantage of this approach is that it is on the one hand implementable with reasonable effort compared to the first discussed implementation alternative because instead of implementing a complete symbolic execution environment from scratch, only a code generator for fUML to, for instance, Java has to be implemented. On the other hand, through a suitable code generation it is possible to generate path conditions through the generated program that correspond to the actual path conditions through the fUML-based behavioral semantics specification. Thus, in general less inputs may be generated and used in the semantic model differencing when applying this approach compared to the second discussed implementation alternative. However, the major drawback of this implementation approach is that the behavioral semantics of fUML has to be redefined by means of a code generator, Thereby, care has to be taken in the implementation of the code generator as it has to be ensured that the produced code conforms to the standardized behavioral semantics of fUML.

Constraint Solving

The result of the symbolic execution of an fUML-based behavioral semantics specification is an execution tree, which consists of path conditions representing all possible execution paths through the semantics specification for a given model. The path conditions are quantifier-free first-order formulas over symbolic values that represent the inputs leading to the execution of the respective path. Thus, each path through the execution tree, that is the conjunction of all path conditions lying on that execution tree path, defines the characteristics of inputs that cause the execution of the respective path of the fUML-based behavioral semantics specification when executing the model on this input. For generating concrete inputs, each of these conjunctions of path conditions has to be solved resulting in an assignment of concrete values to symbolic values. Furthermore, from these computed value assignments, a valid input in terms of objects conforming to the respective modeling language has to be generated.

Solving path conditions constitutes a Boolean satisfiability problem (SAT). Thus, existing SAT solvers may be utilized for solving path conditions. However, making use of them for generating inputs requires the transformation of the fUML-based semantics specification—in particular, of the initialization classes defining the structure of inputs—as well as of path conditions into Boolean variables and Boolean expressions as well as the transformation of satisfying Boolean variable assignments calculated by the SAT solver into objects being instances of the initialization classes. For achieving both, we implemented an integration of fUML with the *model validator plugin* for USE developed by Kuhlmann *et al.* [80].

USE [56] is a UML environment supporting several types of UML diagrams as well as the validation of OCL constraints. The model validator plugin [80] adds the capability to generate UML object diagram being valid instances of a given UML class diagram and fulfilling a given set of OCL constraints. For utilizing the model validator plugin for generating inputs out of path conditions, we implemented several transformations, namely a transformation of Ecore-based metamodels and fUML-based behavioral semantics specifications into UML class diagrams, a transformation of models into UML object diagrams, as well as transformation of path conditions into OCL constraints. The obtained UML class diagrams, UML object diagrams, and OCL invariants are then provided to the model validator plugin of USE. The model validator plugin internally translates the UML diagrams and OCL invariants into the relational logics of

Kodkod [159] to apply an efficient SAT-based search to a UML object diagram that satisfies the UML class diagrams as well as the OCL constraints. The obtained UML object diagram is then translated into a valid instance of the considered modeling language thanks to an additionally implemented model transformation. The obtained instance constitutes an input leading to the execution path represented by the solved path conditions.

6.6 Summary

Managing the evolution of models requires techniques for identifying differences among independently developed or consecutive versions of models. Unlike syntactic model differencing, semantic model differencing does not only consider syntactic differences among two models, such as deletions, additions, and modifications of model elements, but also takes the semantics of the compared models into account. Therewith, semantic model differencing enables further analyses of differences among models, such as analyses of the semantic preservation of changes, and provides the basis for reasoning about the semantics of a change.

In this chapter, we presented a generic semantic model differencing framework that, in contrast to existing semantic model differencing approaches, makes use of the behavioral semantics specifications of modeling languages for supporting the semantic differencing of conforming models. Therewith, our approach follows the spirit of generic syntactic model differencing, which utilized metamodels to obtain the necessary information on the syntactic structure of the models to be compared. By utilizing the behavioral semantics of a modeling language, non-trivial transformations into a semantic domain specifically for enabling semantic differencing can be avoided. Instead, the behavioral semantics specifications of a modeling language, which may also be employed for model simulation, verification, and validation, is utilized to enable semantic model differencing.

Our generic framework performs semantic model differencing by extracting semantic interpretations of the models to be compared from the behavioral semantics specification of the employed modeling language. Therefore, the models to be compared are executed based on the behavioral semantics specification and execution traces are obtained from the execution. The obtained execution traces constitute semantic interpretations over the compared models and can, hence, be used to identify semantics differences among the models. Therefore, the execution traces are syntactically compared by applying match rules that define, based on a suitable semantic equivalence criterion, which differences among the execution traces constitute semantic differences among the models. If such semantic differences are identified, the respective execution trace constitutes a diff witnesses being a piece of evidence about the semantic differences.

In this chapter, we have first shown an instantiation of our semantic model differencing framework for fUML-based behavioral semantics specifications and subsequently discussed how the framework can be generalized to operationally defined behavioral semantics specifications through the introduction of a generic execution trace format. Because our framework performs semantic model differencing on concrete execution traces, inputs for executing the models relevant to the semantic differencing have to be provided as input to the framework. To automate the selection of relevant inputs, we have proposed an approach for generating inputs from fUML-based behavioral semantics specification by applying the idea of symbolic execution.

Semantic differencing for fUML-based semantics specifications. Instantiating our semantic model differencing framework for a concrete semantics specification approach requires that the respective approach provides means for executing models as well as for capturing execution traces reflecting the models' semantics. Our fUML-based behavioral semantics specification approach introduced in Chapter 5 is accompanied with a model execution environment built on top of fUML's execution environment as well as our extensions of this environment. Thus, it enables executing models according to fUML-based behavioral semantics specifications and capturing execution traces in terms of trace models. Thus, our generic semantic model differencing framework can be directly instantiated for fUML-based behavioral semantics specifications. In this instantiation, the semantic model differencing is realized as comparison of trace models obtained from executing the models to be compared.

Semantic differencing for operationally defined semantics specifications. To enable the instantiation of our generic semantic model differencing framework for arbitrary operational semantics approaches, we defined a generic execution trace format. This generic execution trace format constitutes the interface to our framework, which realizes the semantic model differencing as comparison of execution traces adhering to this generic format. Thus, instantiating the framework for a concrete operational semantics approach requires that execution traces conforming to this trace format are captured during the execution of models. We have shown for three distinct operational semantics approaches, how this is possible either through the utilization of the execution environment provided by the respective approach or through the direct introduction of trace capturing functionality into behavioral semantics specifications.

Input generation for fUML-based semantics specifications. Considering all possible inputs to models in the semantic model differencing is infeasible as the number of possible inputs is in general infinite. For automating the selection of inputs relevant to semantic model differencing, we proposed the application of symbolic execution for fUML. Using a combination of symbolic and concrete execution, it is possible to derive all inputs leading to distinct execution traces. These generated inputs may then be used to perform the semantic model differencing. For realizing symbolic execution for fUML, we have presented three distinct implementation alternatives.

We evaluated the expressive power provided by our semantic model differencing framework for defining dedicated semantic differencing operators for modeling languages by implementing existing operators with our framework. Furthermore, we evaluated the performance of the developed semantic model differencing operators. This evaluation is discussed in Section 7.3.

6.7 Related Work

Most of the existing model differencing approaches compare two models based on their *abstract syntax* representation. Thereby, a match between two models is computed yielding correspondences between their model elements, before a fine-grained comparison of all corresponding

model elements is performed. The result of this syntactic differencing is the set of model elements present in only one model and a description of differences among model elements present in both models and corresponding to each other. Thereby, the syntactic differences among two models are usually expressed in terms of edit operations, such as add, delete, and update operations.

One of the earliest works on model differencing was done by Alanen and Porres [3], who presented a metamodel-independent algorithm for differencing UML models based on unique identifiers of model elements. Similar approaches also relying on unique identifiers were proposed by Oliveira *et al.* [125] and Ohst *et al.* [123]. To accommodate scenarios where no unique identifiers are available for finding corresponding model elements, UMLDiff [165] proposes the use of similarity metrics based on the model elements' names and structure for syntactic matching, before the fine-grained differences are computed. Whereas these approaches are explicitly targeted at differencing UML models, other approaches are not restricted to a particular language, such as DSMDiff [90] and EMF Compare [19]. These approaches enable the processing of models conforming to any modeling language by incorporating the modeling language's metamodel into the differencing algorithm to reason about the structure of the models to be compared.

However, to determine whether two syntactically different models also differ in their meaning [63], the semantics of the modeling language they conform to has to be taken into account. Few *semantic model differencing* approaches have been proposed in the past. Generally, we can distinguish enumerative and non-enumerative semantic model differencing approaches [48]. *Enumerative* approaches calculate semantic interpretations of two compared models called *diff witnesses*, which are only valid for one of the two models and, hence, provide evidence about the existence of semantic differences among the models. *Non-enumerative* approaches do not calculate and enumerate diff witnesses directly, but instead compute an aggregated description of the semantic difference among the compared models. The advantage of enumerative approaches is that they directly provide a set of diff witnesses being manifestations of semantic differences among two models. However, because this set might be infinite, enumerative approaches are not complete. In contrast, the advantage of non-enumerative approaches is that they are complete and usually more efficient. However, computing diff witnesses from aggregated difference descriptions computed by non-enumerative approaches is as complex as enumerative approaches.

Maoz *et al.* propose an approach for developing enumerative semantic model differencing operators [94]. In this approach, a semantic model differencing operator is defined through a translation of models into a semantic domain suitable for expressing the semantics of the models as well as for calculating diff witnesses. Diff witnesses obtained in the semantic domain are then translated back to the modeling language. Maoz *et al.* applied this approach for developing the semantic model differencing operators CDDiff [96] and ADDiff [95] for UML class diagrams and UML activity diagrams, respectively. CDDiff translates two UML class diagrams to be compared into one joint Alloy module, which defines predicates representing the semantics of the two UML class diagrams. Furthermore, one additional so-called *diff predicate* is added to the Alloy module, which specifies that all predicates representing the first UML class diagram have to hold and that at least one of the predicates representing the second UML class diagram must not hold. Using the Alloy Analyzer, instances of this diff predicate are calculated, which

constitute diff witnesses as they represent UML object diagrams being instances of the first UML class diagram but not of the second UML class diagram. The obtained instances are then translated back to actual UML object diagrams. ADDiff translates two UML activity diagrams to be compared into SMV modules representing the semantics of the two UML activity diagrams. By applying algorithms traversing the state spaces of the UML activity diagrams, execution traces are identified, which are possible only in one of the two UML activity diagrams and, hence, constitute diff witnesses. The obtained execution traces are then translated back to execution traces of the compared UML activity diagrams.

Unlike Maoz *et al.*, Fahrenberg *et al.* [48] propose an approach for defining non-enumerative semantic model differencing operators. Therefore, the models to be compared are mapped into a semantic domain having an *algebraic structure* that enables the definition of the semantic difference among two models in the form of an operator on that algebraic structure. The semantic difference among two models is captured in terms of a model conforming to the same modeling language as the two compared models. Fahrenberg *et al.* applied this approach for defining semantic model differencing operators for feature models as well as automata specifications [48], and later also for UML class diagrams [47]. The used semantic domains for realizing these differencing operators are Boolean logics, behavioral component algebras, and set theory, respectively. For UML class diagrams, the semantic domain of set theory is used and the semantic model differencing operator is defined as adjoint to the conjunctive merge. Thereby, the semantic difference among two class diagrams is again a class diagram.

While the approaches of Maoz *et al.* and Fahrenberg *et al.* are generally applicable for developing semantic model differencing operators for any modeling language, they have to be regarded as general recipes of how to develop these operators. Following these recipes requires the definition of a modeling language's semantics through the implementation of a translation into a semantic domain in which then specific semantic comparison algorithms and operators have to be defined that perform the semantic differencing. In contrast, our approach is generic in the sense that semantic model differencing is performed by utilizing the behavioral semantics specification of the used modeling language. Therefore, we provide a generic framework that can be instantiated for any operational semantics approach that provides the capability to execute models and capture execution traces adhering to a generic execution trace format. Defining a semantic model differencing operator for a specific modeling language requires only to define match rules comparing execution traces according to a suitable semantic equivalence criterion.

Gerth *et al.* [54] developed an approach concerned with determining whether two *business process models* are semantically equivalent in respect of trace equivalence. Unlike the approaches discussed so far, Gerth *et al.* do not translate business process models into a semantic domain in which then the semantic comparison is performed, but instead transform them into *normalized process model terms*. Thereby, two syntactically different but semantically equivalent business process models are always transformed into the same normalized process model terms. Thus, by syntactically comparing two normalized process model terms obtained from the transformation of two business process models, it can be determined whether these business process models are semantically equivalent. Gerth *et al.* use this approach for detecting semantically equivalent fragments of business process models in order to enhance the detection of conflicting change operations applied to the models [53].

A similar idea underlies the approach proposed by Reiter *et al.* [131] for detecting semantic conflicts among change operations applied to models. In their approach, two compared models are transformed into so-called *semantic views* using model transformation techniques. Thereby, the semantic view of a model filters out any information about the model not relevant to conflict detection. Based on syntactic model differencing techniques applied to the semantic views of two models, semantic conflicts are detected. The underlying assumption of this approach is, that a semantic view enabling the detection of semantic differences among models through syntactic comparison exists and that this semantic view can be generated from a model through a model transformation.

While the approach by Gerth *et al.* is targeted at business process models only also the approach by Reiter *et al.* is restricted concerning its applicability to modeling languages in general. Furthermore, both approaches focus on enhancing conflict detection through the identification of semantic equivalent fragments of models in order to avoid false-positive conflicts.

CHAPTER 7

Evaluation

7.1 Extensions of the fUML Execution Environment

7.1.1 Research Questions

The aim of extending fUML's execution environment was to enable the implementation of methods and techniques for analyzing executable UML models based on the fUML virtual machine. Therefore, we developed an event mechanism, command interface, and trace model, which we integrated with the fUML virtual machine to support in particular the implementation of testing, dynamic analysis, debugging, and non-functional property analysis methods (cf. Chapter 4).

We evaluated the developed extensions of the fUML execution environment concerning adequacy and performance. In particular, we aimed at answering the following research questions.

Research question 1: Adequacy. *Are the developed extensions of the fUML execution environment adequate for implementing testing, dynamic analysis, debugging, and non-functional property analysis methods?*
This research question is divided into the following subquestions.

(a) **Event mechanism.** Are the events issued by the event mechanism and the information they carry adequate for thoroughly observing state changes of fUML model executions?

(b) **Command interface.** Are the commands provided by the command interface adequate for flexibly controlling fUML model executions?

(c) **Trace model.** Are trace models an adequate basis for reasoning about the runtime behavior of fUML model executions?

Research question 2: Performance. *How much performance overhead is caused by the developed extensions of the fUML execution environment?*
This research question is divided into the following subquestions.

(a) **Execution time.** How much does the execution time of an fUML model increase due to issuing events, managing execution state information, and recording trace models?

(b) **Memory consumption.** How much does the memory consumption increase due to the issued events, captured execution state, and created trace model?

In order to answer these research questions, we evaluated the developed extensions of the fUML execution environment by carrying out case studies.

To evaluate whether the extensions are adequate for implementing analysis methods, we implemented based on them dedicated analysis tool prototypes, namely a debugger, testing framework, and performance analysis tool. In Section 7.1.2, we briefly present these prototypes, point out which extensions have been employed for developing the prototypes, and discuss the adequacy of the extensions for implementing the prototypes.

Because the extensions have been integrated with the reference implementation of the fUML virtual machine [91], we evaluated the performance overhead caused by the extensions by comparing the performance of the reference implementation as is (i.e., without extensions) and of the extended reference implementation (i.e., with integrated extensions). We report on this evaluation in Section 7.1.3.

7.1.2 Adequacy

By utilizing our extensions of the fUML execution environment, we prototypically implemented a debugger, testing framework, and performance analysis tool for fUML. These tool prototypes depend heavily on the developed event mechanism, command interface, and trace model as they need to observe, control, and analyze model executions carried out by the fUML virtual machine. Hence, the development of these prototypes served as case studies for qualitatively evaluating the adequacy of our extensions of the fUML execution environment with respect to the implementation of dedicated analysis methods and techniques for executable UML models based on fUML. In the following, we briefly present the capabilities of the tool prototypes and discuss how the extensions of the fUML execution environment have been applied for implementing them.

Case Studies

Debugger. Debugging is a method that requires not only a powerful mechanism for controlling the execution of a program, but also depends heavily on event notifications about state changes of the execution during runtime, and demands for very precise runtime information about the execution. Hence, implementing a debugger for fUML models relies on the ability to control ongoing model executions, as well as on the ability to observe and analyze model executions. Therewith, all three developed extensions of the fUML execution environment have to be applied in order to implement a debugger for fUML models on top of the fUML virtual machine. We implemented such a debugger for fUML models [97] based on the extended fUML execution environment, which provides the capabilities to *control* the execution of a model by offering debug commands, such as *step into* and *resume*, and the possibility to set breakpoints; as well as to *observe* the state of the execution consisting of the current position of the execution and

values of existing variables. The debugger is integrated with the Eclipse Debug Framework and allows debugging activities created with the Papyrus UML editor[1].

Regarding the *control* of executions, a debugger should empower users to stepwise resume a suspended execution using the commands *step into*, *step over*, *step return*, and *resume*. In the context of fUML, *step into* and *step over* should cause the execution of a selected currently enabled activity node. However, if this node is a call action calling an activity, *step into* should cause the execution to suspend directly before the first node of the called activity is executed, whereas *step over* should execute the entire called activity and suspend after the call action itself has been executed. Accordingly, *step return* should execute the entire currently executing activity. This can be realized easily using the developed command interface and event mechanism. Therefore, we call the operation nextStep() of the command interface class ExecutionContext repeatedly until a certain event is issued by the event mechanism. For instance, in the case of *step return*, nextStep() is called until an activity exit event for the currently executing activity is received informing about the completion of this activity execution. Thereafter, the execution suspends, which is reported by a suspend event. The command *resume* as well as the handling of breakpoints are directly supported by the command interface.

To enable the *observation* of the current state of an execution, a debugger usually depicts the state of the program being debugged in terms of threads, stack frames, and variables. A *thread* is a sequence of actions that may execute in parallel with other threads. In the context of fUML, multiple activity nodes may be enabled at the same time, for instance, if the control flow of an activity exhibits fork nodes. As a result, the concurrently enabled activity nodes can be executed separately without affecting the execution of the other enabled nodes. Hence, we consider each concurrently enabled node to run in an own thread. For deriving the currently running threads, we may obtain the enabled nodes of a suspended activity execution easily from the trace model, as it captures for each enabled node an ActivityNodeExecution instance having both attributes underExecution and executed set to false. Alternatively, the enabled nodes could be retrieved from the command interface class ExecutionContext using the operation getEnabledNodes() or from the last received suspension event issued by the event mechanism. However, a new enabled node does not always constitute a new thread, because the node activation may result from resuming an already existing thread. To obtain this information, the trace model can be used, which contains for each enabled node the information about logical predecessors. If an existing thread concerns the logical predecessor of a new enabled node, this thread may be updated to the new enabled node. If it has several predecessors for which threads exist, one of them may be updated and the others are terminated. If we do not find a corresponding thread or all corresponding threads have been assigned to other new enabled nodes already, we create a new thread. A *stack frame* represents the runtime context of a suspended thread, which is in our context the enabled activity node assigned to the respective thread. A stack frame may have child stack frames that represent call actions that called the activity where the assigned node resides in. Thus, for deriving the stack frame hierarchy, we have to be able to find out whether an activity node resides in an activity that itself has been invoked by a call action. Therefore, we again make use of the trace model by navigating from the ActivityNodeExecution instance representing the execution of the activity node upwards to its containing ActivityExecution instance and check

[1] `http://www.eclipse.org/papyrus`, accessed 10.07.2014

whether it refers to a CallActionExecution as caller. If this is the case, we add a child stack frame for this call action. This procedure can be repeated until the upper most ActivityExecution instance captured by the trace model is reached. A *variable* denotes a data value that is in the scope of the current stack frame. As stack frames are associated with activity nodes, variables refer to their inputs. Hence, obtaining the variables is easily possible by looking up the ActivityNode-Execution instance in the trace model representing the execution of the activity node associated with the respective stack frame and retrieving the captured ValueSnapshot instances constituting its inputs. Furthermore, all extensional values residing at the locus of the current execution are in the scope of each stack frame. Consequently, they are associated with each existing stack frame as variable. Therefore, the trace model can be utilized, as it captures state information about all extensional values residing at the locus of execution. Hence, we retrieve all objects and links captured by the trace in terms of ValueInstance instances for whom no destroyer exists yet, and obtain their most current ValueSnapshot. The state of all threads, stack frames, and variables has to be updated when the activity execution suspends, which is indicated by suspension events issued by the event mechanism.

Further debugging capabilities could be easily implemented thanks to the extensions of the fUML execution environment. For instance, *watchpoints* causing the suspension of an ongoing model execution in case a certain extensional value was modified could be realized similar to the debug commands *step into*, *step over*, and *step return* by calling the command interface operation nextStep() until an extensional value event reporting on the modification of this certain extensional value is issued by the event mechanism. Likewise, *state modifications* in terms of altering the values of variables in the scope of a current stack frame could be supported, as the command interface provides access to the execution environment of the fUML virtual machine and therewith to the locus of the execution and all existing extensional values. The very same command interface functionality could also be utilized for implementing *hot code replacement* enabling the modification of a model currently being executed. Furthermore, *reverse debugging* could be realized based on the detailed runtime information provided by the trace model and the ability to access and modify the locus of the execution provided by the command interface.

Testing framework. A commonly applied approach for testing a system is to define test cases that assert whether the system yields expected outputs for defined inputs. Based on asserting input/output relations in this way, it can be validated whether the system fulfills its functional requirements as well as verified whether it contains defects. Testing fUML models by asserting input/output relations is already supported by the fUML virtual machine as is. Therefore, input data for executing an activity to be tested can be provided to the fUML virtual machine and the output of the execution can be retrieved from the fUML virtual machine and compared with the expected output. Hence, developing testing frameworks supporting input/output relation assertions of fUML models is possible based on the fUML virtual machine without the need for any extensions. However, performing a more fine grained analysis of the behavior of an fUML model for testing purposes requires thorough and precise runtime information about its execution and, therewith, requires dynamic analysis capabilities. For achieving this, the trace model developed as fUML execution environment extension can be applied. By applying the trace model, we implemented a testing framework [104, 105] integrated with EMF that not only en-

ables the assertion of input/output relations of activities, but also the runtime state of the model during execution as well as the execution order of model elements. Our testing framework consists of a test specification language and a test interpreter. Using the test specification language the modeler can specify test cases comprising test input data and assertions on the behavior of activities contained by the fUML model under test. The test interpreter executes the activities under test for the defined input data using the extended fUML execution environment and evaluates the defined assertions based on the obtained trace models. The testing framework supports two types of assertions, namely execution order assertions and state assertions.

Execution order assertions can be used to verify whether the nodes contained by tested activities are executed in the correct chronological order. Therefore, execution order assertions are defined by specifying the nodes in the order in which they should be executed. It is also possible to specify the relative order of nodes and omit nodes, whose execution order is irrelevant for the test case. Evaluating execution order assertions is easily possible based on obtained trace models. Therefore, we simply investigate the ActivityNodeExecution instances contained by a trace model, which represent the executed activity nodes, as well as their chronological execution order, which is captured by the references chronologicalPredecessor and chronologicalSuccessor. To evaluate order assertions in case the activities under test contain concurrent branches of activity nodes, the captured logical dependencies between the executed activity nodes represented by the references logicalPredecessor and logicalSuccessor can be investigated to identify all possible sequential orders in which activity nodes contained by concurrent branches can be executed.

State assertions can be used to verify whether the runtime state of a tested models evolves as expected during its execution and, thus, enables the assertion of the correct behavior of models. Thereby, the runtime state of a model at a certain point in time of the execution consists of all values existing at this exact point in time and their state at this point in time. For instance, the state of an object comprises all its feature values and links to other objects. In a state assertion, the runtime states to be checked are specified using temporal expressions. An example of a temporal expression is *always after actionX* selecting each runtime state of the model after *actionX* has been executed. The actual assertion of the selected runtime states is defined using OCL expressions. For evaluating state assertions, the possibility to retrieve the runtime state of the tested model at any point in time of the execution is required. This possibility is provided by the trace model, as it captures all modifications of values including their creation and destruction by ValueInstance and ValueSnapshot instances. As the trace model captures for each ValueSnapshot instance the information about which action execution, represented as ActionExecution instance, led to the respective value modification, as well as the information about the chronological execution order of actions, it can be determined which values existed at a certain point in time of the execution as well as their states at this point in time. Thus, it is possible to derive the runtime state of the model at any point in time of the execution and, hence, evaluate temporal expressions of state assertions. The evaluation of OCL expressions on these runtime states, i.e., on the values existing in the respective state, is then achieved using the Dresden OCL framework [65].

Based on our trace model, further testing capabilities could be implemented on top of the fUML virtual machine. For instance, *test coverage metrics* could be easily calculated based on the captured information about executed activities and activity nodes. Furthermore, by utilizing the information about logical dependencies between executed activity nodes the *location*

of defects leading to failing assertions could be determined and *corrective feedback* could be calculated supporting the modeler in correcting defects.

Non-functional property analysis tool. To perform model-based analyses of non-functional system properties, models defining the structure and the behavior of the system as well as factors influencing the respective non-functional property are analyzed. Therefore, either analytical methods or simulation techniques are employed. Applying simulation techniques requires detailed runtime information about the execution of the model used for the analysis. We proposed to employ the trace model developed as fUML execution environment extension to implement simulation techniques for analyzing non-functional properties based on fUML models [10]. Doing so enabled us to implement two performance analysis tools for fUML models. The first one can be used to calculate the end-to-end execution time of component-based software systems [10]. The second one also allows the computation of additional performance indices, such as response time, throughput, and utilization, by taking the contention of resources into account [49].

To analyze the *end-to-end execution time* of a component-based software system, an fUML model defining the software architecture and the hardware platform of the system, as well as performance-related information is processed. The definition of the software architecture comprises the definition of the system's components in terms of classes as well as the definition of the components' behavior in terms of operations and associated activities. The hardware platform is defined in terms of classes specifying the execution hosts of the system. The software architecture and the hardware platform are connected with each other through dedicated associations defining the deployment of software components on execution hosts. Performance-related information required for analyzing the system's performance is defined by applying stereotypes of the UML profile MARTE [115] to the model. MARTE is a UML profile primarily targeted at supporting the modeling and analysis of real time and embedded systems. However, it provides stereotypes for annotating UML models with information required for performance and schedulability analysis, which are also useful for the analysis of systems other than real time and embedded system. We use several of these stereotypes to specify the characteristics of the execution hosts defined in an fUML model of a component-based software system, such as used hardware resources like CPU, memory, and network connection. Furthermore, the resource usage of the software components for executing a certain behavior is defined by applying the MARTE stereotype *ResourceUsage* to the activity defining the respective behavior. Lastly, a scenario, whose end-to-end execution time shall be analyzed, has to be defined in terms of an activity invoking software component operations. This scenario can be simulated by executing the defined activity using the fUML virtual machine. For computing the required end-to-end execution time, the information about which activities have been executed and how often they have been executed is required. This can be easily retrieved from the trace model, as each activity execution is represented by an ActivityExecution instance. By combining this information and the information about the resource usage of the executed activities defined via stereotype applications, the end-to-end execution time can be calculated.

For analyzing the performance of a system taking *resource contention* into account, additional information has to be specified in the fUML model defining the system to be analyzed.

In particular, workloads have to be specified defining how often a certain scenario has to be processed by the system in a given time frame, i.e., how often a job requiring the execution of this scenario arrives at the system. Therefore, the MARTE stereotype *GaScenario* is applied to each activity defining a scenario and job arrival patterns are defined. For calculating waiting times of jobs caused by resource contention, information about which components are involved in processing the jobs is required. Therefore, the activity defining the scenario that has to be executed for processing a job can be executed using the fUML virtual machine. The captured trace model provides the information about which activities have been executed and, therewith, the information about which components are involved in processing the respective job. Based on this information and the calculation of the execution time needed by each involved component for processing a job as described beforehand, it is possible to compute the waiting times of jobs in the system. In further consequence, the performance indices response time, throughput, and utilization can be calculated.

By combining the detailed information about the runtime behavior of an fUML model captured by trace models, as well as information about factors influencing non-functional system properties defined in the fUML model itself or via stereotype applications, it is possible to implement various kinds of further non-functional property analysis methods.

Results

Table 7.1 summarizes the applications of the developed extensions of the fUML execution environment for implementing the presented debugger, testing framework, and performance analysis tools.

The trace events issued by the event mechanism as well as the execution control provided by the command interface enabled us to implement the traditional debug commands *step into*, *step over*, *step return*, and *resume*, as well as breakpoint support for debugging fUML models. Based on the runtime information provided by the trace model about the execution of activities and actions, their inputs and outputs, as well as their logical dependencies through the sending and receiving of tokens, we could implement support for observing the state of executions as usually provided by IDEs in terms of threads, stack frames, and variables. Through suspension events, the debugger implementation detects the suspension of executions and, therewith, the need for updating the provided state information. As the trace model is continuously updated during the model execution, up to date runtime information about running threads and their associated stack frames and variables can be provided to users.

The runtime information about executed activities and activity nodes, as well as about logical dependencies between them, which is captured by the trace model, allowed us evaluate execution order assertions. They provide users with the means for testing the correct execution order of activities and activity nodes. The runtime information about executed actions as well as their inputs and outputs is sufficient for reconstructing each runtime state of a carried out model execution comprising the state of all existing values at a certain point in time of the execution. This enabled us to evaluate state assertions providing users with the means to test the correct behavior of fUML models.

For implementing performance analysis methods, which are based on simulation techniques, we needed to combine information about the executed parts of a simulated model with perfor-

	Event mechanism	Command interface	Trace model
Debugger			
Control via debug commands and breakpoints	Trace events	Execution control	-
Observation of position and variable values	Trace events	-	Executions Inputs and outputs Token flows
Testing framework			
Execution order assertions	-	-	Executions Token flows
State assertions	-	-	Executions Inputs and outputs
Performance analyzer			
Execution time analysis	-	-	Executions
Resource contention analysis	-	-	Executions

Table 7.1: Applications of fUML execution environment extensions in case studies

mance characteristics of the modeled system. While the latter information is usually captured by the application of MARTE stereotypes, the former information has to be retrieved from the simulation environment—in our case the fUML execution environment. The runtime information provided by the trace model proved to provide sufficient information for this purpose.

Besides the debugging, testing, and performance analysis capabilities implemented in the course of the reported case studies, we also identified further capabilities that could be implemented based on the developed extensions of the fUML execution environment, such as watchpoints for debugging and defect location for testing.

From developing the presented tool prototypes, we conclude that our extensions of the fUML execution environment are adequate for implementing debugging, testing, dynamic analysis, and non-functional property analysis tools. In particular, we argue that the event mechanism is adequate for observing state changes of model executions during runtime on a fine granularity level, such as required for debugging. In combination with trace models captured for model executions, it is possible to perform detailed dynamic analyses of the runtime behavior of models either during their execution or a posteriori, for instance for testing or performance analysis purposes. Furthermore, the command interface allows the flexible control of the execution of models as for example needed for debugging.

By developing the presented tool prototypes, we also identified potential for improving our extensions of the fUML execution environment discussed in the following.

One potential for improvement concerns the developed event mechanism, namely its information content in terms of the completeness of the runtime information observable through events. Currently, two types of events are issued by the event mechanism, namely trace events for determining the current position of an ongoing execution (i.e, the activity started in the last execution step, or the action or control node executed in the last execution step) and extensional

value events for determining modifications of extensional values. However, the events do not carry information about inputs and outputs of executing activities, actions, and control nodes, information about token flows, as well as information about logical dependencies between actions and control nodes. This information can currently be only retrieved from recorded trace models. Thus, the implemented debugger prototype has to utilize the trace model for obtaining the runtime information required for displaying the current state of the executing model, instead of relying on the event mechanism and command interface alone. Extending the event mechanism with additional event types carrying this runtime information would enable relying on the event mechanism only for observing the state of an executing model. Furthermore, it would enable the replacement of the current mechanism for recording the execution state, which is part of the command interface, with an execution event listener and the decoupling of the trace model capturing from the execution state by realizing the trace capturing component also as an execution event listener.

The command interface allows the execution of a model in a stepwise manner. Thereby, an execution step comprises the start of executing an activity, the completion of executing an action, or the completion of executing a control node. However, it does not consider the consuming of tokens as well as the sending of token offers through activity edges as an execution step, which might be of interest for debugging an fUML model on a more detailed level.

Concerning the trace model, we experienced that one runtime information that is often required is the runtime state of the model at a certain point in time of the execution consisting of all values existing at this point in time and their state at this point in time. This is required by the developed testing framework for evaluating state assertions and would be also beneficial for the developed debugger for displaying the current state of an executing model. However, this information is already captured by trace models and could be easily represented in a more direct way requiring only minor adaptations of the trace metamodel.

One limitation of all extensions concerns the completeness of the support of fUML. In particular, signals and active classes are currently not supported including the action types send signal action, accept event action, start classifier behavior action, and start object behavior action. Furthermore, we currently do not support the definition of activities as reducer behavior of reduce actions and as decision input behavior of decision nodes. However, the former can be replaced by the usage of expansion regions and call behavior actions, and the latter by the usage of call behavior actions and subsequent decision nodes without decision input behavior.

7.1.3 Performance

The extensions of the fUML execution environment have been integrated with the reference implementation of the fUML virtual machine [91]. For detecting state changes of model executions being carried out by fUML virtual machine, interrupting model executions, and capturing state information of model executions, which is required for the realization of the event mechanism, command interface, and trace model, respectively, we implemented dedicated aspects using AspectJ and wove these aspects into the reference implementation. For quantitatively evaluating the performance overhead caused by the developed extensions, we carried out a case study in which we compared the performance of the extended fUML execution environment with the standard fUML execution environment, i.e., with the reference implementation of the

fUML virtual machine. In the following, we present the performed case study and discuss the evaluation results.

Case Study

For comparing the performance of the extended fUML execution environment and the standard fUML execution environment, we developed an fUML model, which defines the structure and the behavior of an *online store* and serves as case study example.

Figure 7.1 depicts the classes defining the structure of the online store. At the top of this figure, the classes defining the *entities* managed by the online store are shown. At the bottom of this figure, the classes defining the *services* provided by the online store are shown. The online store manages registered customers (class *Customer*), a catalog of products (classes *Product* and *Item*), shopping carts of customers (classes *Cart* and *CartItem*), as well as orders placed by customers (classes *Order* and *OrderLine*). Services are offered by the *ApplicationController* of the online store, which is responsible for handling user interactions. Therefore, it uses three other software services, namely *CustomerService*, *CatalogService*, and *OrderService*, that manage the customers, products, and orders of the online store, respectively. These services, in turn, have access to an *EntityManager* that provides operations to persist, retrieve, and delete entity objects.

Figure 7.2 shows three usage scenarios of the online store. In the *login scenario*, a user logs on to the online store by calling the operation *login()* of the *ApplicationController* and providing login credentials. The login is handled by the *CustomerService* who retrieves the *Customer* object corresponding to the login credentials from the *EntityManager*. In the *find item scenario*, a user retrieves an item from the product catalog of the online store. Therefore, the operation *findItem()* of the *ApplicationController* is called and the name of the item to be retrieved is provided as input to this operation. For retrieving the respective item, the *ApplicationController* calls the operation *findItem()* of the *CatalogService*, which in turn calls the operation *findAll-Items()* of the *EntityManager* and iterates over the obtained items for retrieving the item with the provided name. In the *buy scenario*, the user first logs on to the online store as defined by the *login scenario*, retrieves then an item from the product catalog as defined by the *find item scenario*, adds the retrieved item to the cart, and finally orders the item. For adding the retrieved item to the cart, the operation *addItemToCart()* of the *ApplicationController* is called, which forwards the request to the *OrderService* for adding the provided item to the cart and persisting the updated cart using the *EntityManager*. For ordering the item, the operation *confirmOrder()* of the *ApplicationController* is responsible. It calls the operation *confirmOrder()* of the *Order-Service*, which creates a new order and persists the order as well as deletes the cart by using the *EntityManager*. Each usage scenario is modeled as a UML activity, which calls the respective operations of the service class *ApplicationController*. Beforehand, two *Customer* objects, three *Item* objects, and one object of each service class are created and appropriately initialized.

Setup

The performance comparison was done by executing the activities defining the three usage scenarios *login scenario*, *find item scenario*, and *buy scenario* of the online store. Thereby, each activity was executed by means of the the standard fUML execution environment as well as

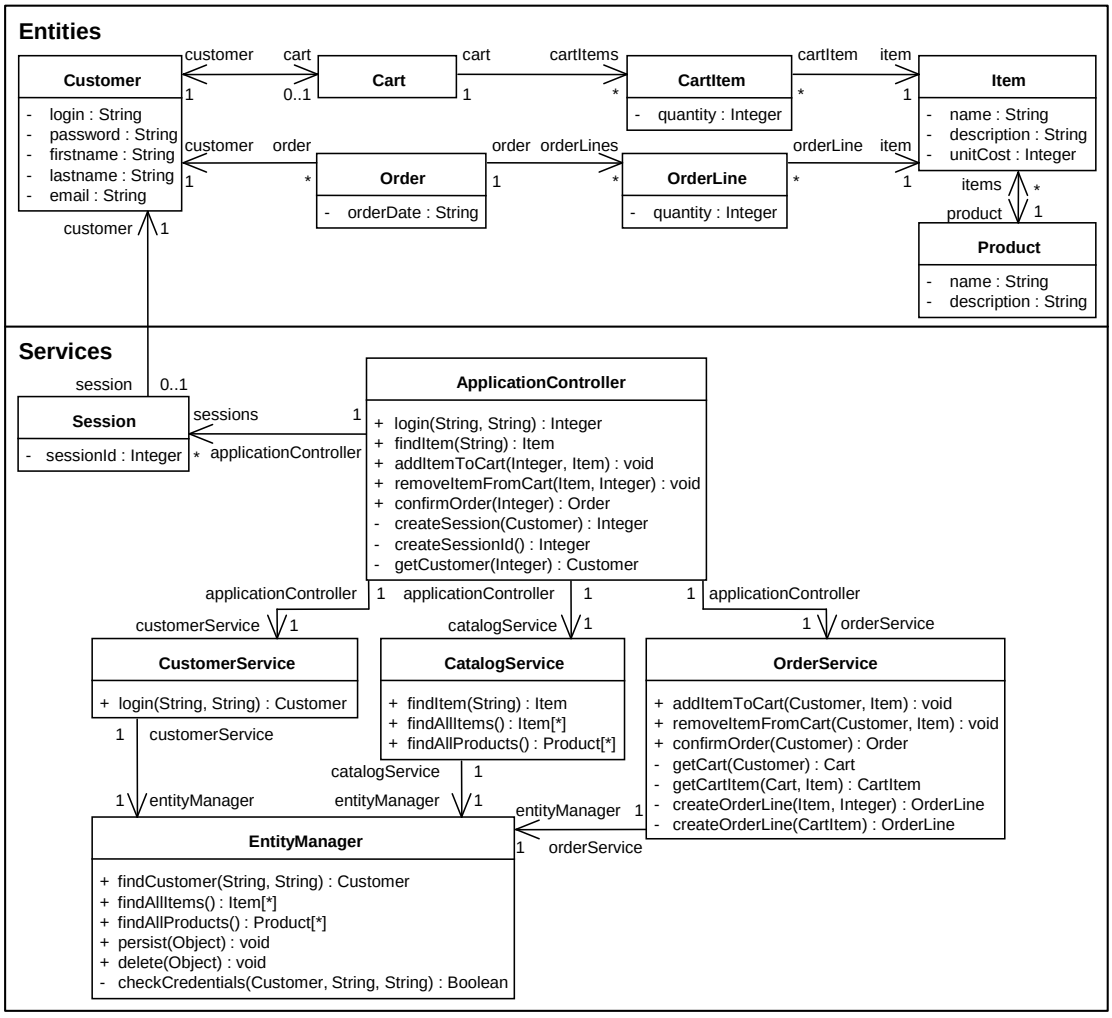


Figure 7.1: Online store case study: Classes

the extended fUML execution environment. For each activity, we measured the *execution time* needed by the respective fUML execution environment for executing it, as well as the respective *memory consumption*.

The execution time was measured by taking timestamps right before the model execution was started and right after the model execution was finished. For the standard fUML execution environment, timestamps were taken right before calling the operation execute() of the execution environment class Executor responsible for synchronously starting the execution of an activity, and right after this operation finished execution. For the extended fUML execution environment, timestamps were taken right before calling the operation executeStepwise() of the command interface class ExecutionContext responsible for stepwise executing an activity, and right after the last execution step finished. This enabled us to compute the additionally needed execution

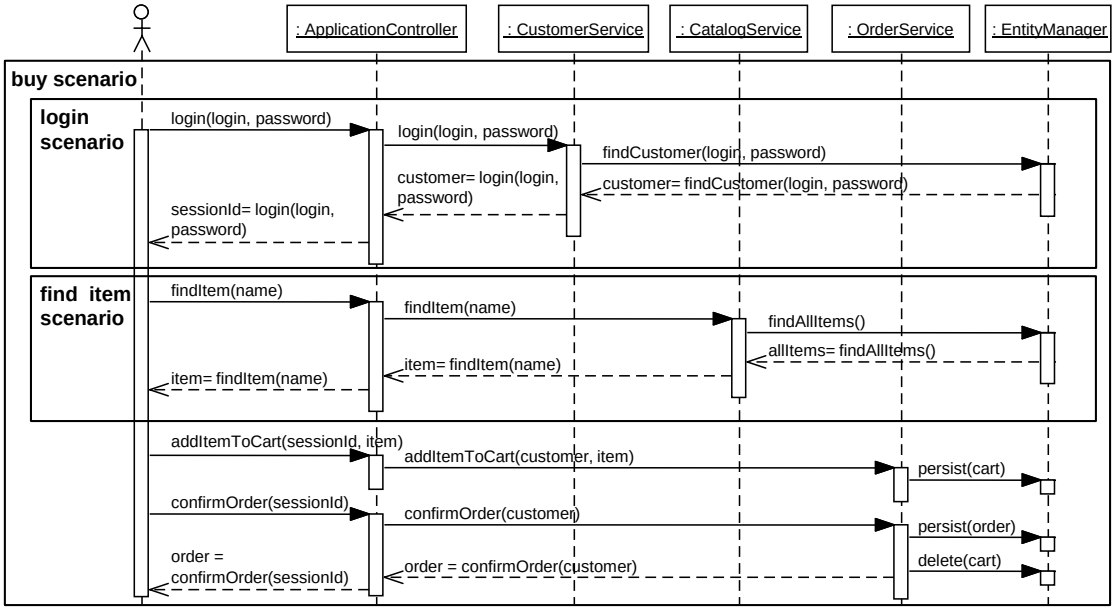


Figure 7.2: Online store case study: Scenarios

time induced by the event mechanism, command interface, and trace model recording.

The memory consumption was measured by recording the objects allocated on the heap during executing the respective activity using JProfiler[2]. This enabled us to measure the number and size of objects allocated on the heap, which are relevant to the evaluation, that are instances being model elements, execution model elements, events, execution state objects, or trace model elements. Thus, we could calculate the additional memory consumption caused by the events issued by the event mechanism, execution state information kept by the command interface, and recorded trace models.

In the following, we explicate the hardware and software environment used for carrying out the performance evaluation.

- **Hardware:** Intel Dual Core i5-2520M CPU 2.5 GHz, 8 GB RAM
- **Operating system:** Windows 7 Professional, Service Pack 1
- **Eclipse:** Kepler, Service Release 1, Build 20130919-0819
- **JProfiler:** Version 8.0.7, Build 8077
- **Java:** Version 7, Update 3, Build 1.7.0_03-b05

[2] `http://www.ej-technologies.com`, accessed 16.07.2014

Results

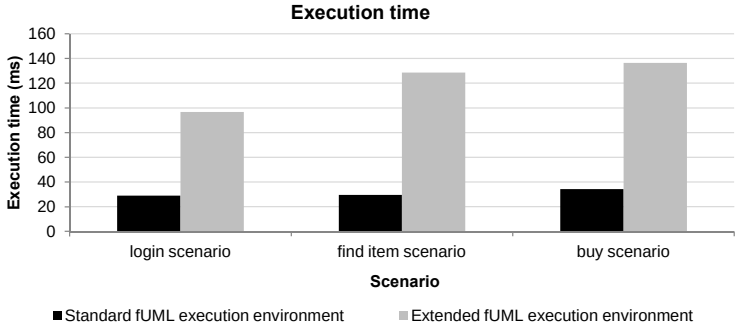
Execution time. Figure 7.3 shows the execution times measured for the standard fUML execution environment as well as the extended fUML execution environment. We executed each usage scenario 20 times using both environments, measured the execution times, and calculated their arithmetic mean (AM) as well as standard deviation (STD) (cf. table at the bottom of Figure 7.3). For the *login scenario*, the extended fUML execution environment was slower by a factor of 3, for the *find item scenario* and the *buy item scenario* by a factor of 4. From these results, we conclude that our extensions of the fUML execution environment seriously compromise the performance of the fUML virtual machine.

To find the cause for these performance results, we further analyzed the execution times required for executing the individual operations comprising our extensions based on our case study example using the Eclipse plugin JVM Monitor[3]. The execution time required for executing operations associated with recording trace models accounts for around 40% of the overall execution time concerning the extensions. The execution of operations associated with querying and updating execution state information captured by the command interface accounts for around 20%. The execution control operations provided by the command interface account for around 34%. The remaining 6% are concerned with creating and issuing events by the event mechanism. From this analysis, we conclude that there are two main sources of the performance overhead introduced by our extensions of the fUML execution environment: (i) Because we did not modify the fUML execution environment directly but used aspect-oriented programming techniques, we had to collect much execution state information using dedicated aspects in order to realize the execution control and trace capturing capability provided by the command interface and trace model, respectively. (ii) Because the runtime information captured by trace models are very detailed in order to provide high precession, recording trace models during ongoing model executions is expensive. To mitigate the performance overhead introduced by our extensions, we recommend to redesign the fUML execution environment considering the integration of means for observability, controllability, and dynamic analyzability as proposed by our event mechanism, command interface, and trace model. Furthermore, we expect that providing means for configuring the level of detail of runtime information captured by trace models would result in performance improvements in cases were only parts of the runtime information are required.

Memory consumption. Figure 7.4 shows the results of the memory consumption measurements. It depicts the size of the objects allocated on the heap for executing the three usage scenarios of our online store case study. It is distinguished between model elements, execution model elements, events issued by the event mechanism, execution state objects kept by the command interface, and trace model elements. The objects of the latter three types constitute the overhead of the extensions incorporated into the fUML execution environment in terms of memory consumption. It amounts to 57 kB, 36 kB, and 116 kB compared to 5,454 kB, 5,403 kB, and 5,614 kB allocated for the model and the execution model. Thus, the measured memory consumption overhead lies between 0.66 % and 2.03%. We measured a similar overhead in terms of number of objects. From these results, we conclude that our extensions of the fUML execu-

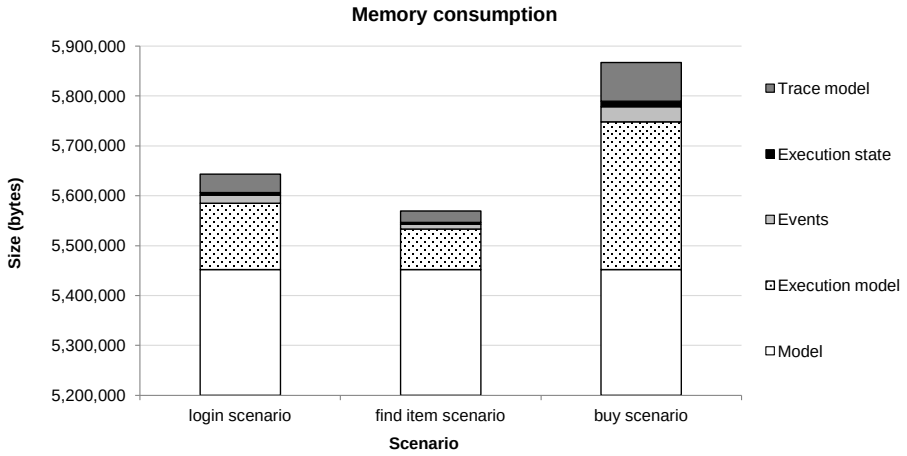
[3] `http://www.jvmmonitor.org`, accessed 17.07.2014

tion environment cause only a marginal memory overhead. This can be justified by the fact that the objects instantiated by the extensions act mainly as references to model elements but do not carry much data. For instance, trace events issued by the event mechanism refer to activities and activity nodes, whose execution started and finished, but do not carry any information except for a time stamp of the data type Long and an identifier of the type Integer. Nevertheless, the number of issued events and the size of captured trace models grow linear with the number of executed model elements and the number of performed value modifications.



Scenario	Standard fUML execution environment		Extended fUML execution environment	
	AM	STD	AM	STD
login scenario	29 ms	21 ms	97 ms	46 ms
find item scenario	30 ms	25 ms	129 ms	77 ms
buy scenario	34 ms	24 ms	136 ms	78 ms

Figure 7.3: Execution time measurements for online store case study



Scenario	Model	Execution model	Events	Execution state	Trace model
login scenario	5,452,440 (96.62%)	132,328 (02.34%)	16,152 (00.29%)	5,168 (00.09%)	37,112 (00.66%)
find item scenario	5,452,440 (97.89%)	80,520 (01.45%)	10,136 (00.18%)	3,360 (00.06%)	23,384 (00.42%)
buy scenario	5,452,440 (92.93%)	296,024 (05.05%)	29,976 (00.51%)	10,944 (00.19%)	78,120 (01.33%)

Figure 7.4: Memory consumption measurements for online store case study

7.2 Semantics Specification with fUML

7.2.1 Research Questions

With the integration of fUML into existing metamodeling languages, metamodeling methodologies, and metamodeling environments, we aimed at providing the means for defining executable modeling languages with the standardized UML 2 compliant action language of fUML. Therefore, we elaborated a language integration strategy for integrating fUML with existing metamodeling languages resulting in the executable metamodeling language xMOF, as well as a methodology for developing executable semantics specifications with this language. We aimed at constructing the language integration strategy and semantics specification methodology in a non-invasive way, meaning that existing metamodeling languages and techniques, as well as tools provided by existing metamodeling environments for defining modeling languages and processing models should remain unaffected. This includes also, that an existing definition of a modeling language in terms of a metamodel should not have to be modified in order to define the language's behavioral semantics. Instead, the definition of the behavioral semantics of modeling concepts should build upon a modeling language's metamodel but be clearly separated from it.

To investigate the fulfillment of these requirements we evaluated the adequacy of fUML and the elaborated semantics specification methodology for developing executable modeling languages, as well as the non-invasiveness of the developed language integration strategy and semantics specification methodology. In particular, the evaluation investigated the following research questions.

Research question 1: Adequacy of fUML as a semantics specification language. *Is fUML adequate for formally defining the behavioral semantics of executable modeling languages?*
This research question is divided into the following subquestions.

(a) **Expressiveness.** Is fUML expressive enough to formally define the behavioral semantics of modeling languages such that conforming models can be executed?

(b) **Suitability.** Is fUML suitable as semantics specification language?

Research question 2: Adequacy of the semantics specification methodology. *Is the semantics specification methodology adequate for developing behavioral semantics specifications with fUML?*
This research question is divided into the following subquestions.

(a) **Adequacy for semantics specification development.** Does the semantics specification methodology allow the systematic and efficient development of behavioral semantics specifications with fUML?

(b) **Adequacy for model execution.** Does the semantics specification methodology allow the direct execution of models based on developed behavioral semantics specifications?

(c) **Adequacy for analysis of semantics specifications.** Does the semantics specification methodology allow the utilization of runtime information obtained from performing model executions to analyze developed behavioral semantics specifications?

Research question 3: Non-invasiveness of fUML integration. *Is the integration of fUML with metamodeling languages, metamodeling methodologies, and metamodeling environments non-invasive?*
This research question is divided into the following subquestions.

(a) **Non-invasiveness of language integration.** Is the language integration strategy non-invasive in the sense that the definition of metamodeling languages by meta-metamodels as well as any techniques and tools based on meta-metamodels remain unaffected by the integration?

(b) **Non-invasiveness of semantics specification methodology.** Is the semantics specification methodology non-invasive in the sense that metamodeling methodologies and metamodeling environments remain unaffected such that their processes, techniques, and supporting tools remain applicable as before?

(c) **Non-invasiveness of semantics specification.** Is the definition of a modeling language's behavioral semantics developed with fUML and the elaborated methodology non-invasive in the sense that both the metamodel of the modeling language and any existing tool support for the modeling language remain unaffected?

We instantiated the elaborated language integration strategy and semantics specification methodology for EMF by accordingly integrating fUML with Ecore as well as implementing corresponding tool support for EMF. Using these implementations of xMOF and our semantics specification methodology, we carried out five case studies, in which we developed the behavioral semantics of five distinct modeling languages of different complexities and characteristics [86, 100]. By drawing conclusions from these case studies we answered the posed research questions.

7.2.2 Case Studies

For answering the above stated research questions, we developed the behavioral semantics specifications of five distinct modeling languages. Therefore, we used our implementation of xMOF for Ecore and followed the semantics specification methodology accompanying xMOF by utilizing the tool support implemented for EMF. The considered modeling languages are (i) Petri nets, (ii) a modeling language for imperatively defining computations, (iii) finite state automata, (iv) UML activity diagrams, and (v) UML class diagrams. The criteria applied for selecting these modeling languages are the diversity of *semantic paradigms* underlying the modeling languages as well as their *complexity* in terms of the complexity of the modeling languages' metamodels. In the following, we discuss the modeling concepts provided by the considered modeling languages as well as their behavioral semantics specifications developed in the course of the case studies. Thereafter, we summarized the characteristics of these modeling languages and provide insights into the complexity of the developed behavioral semantics specifications.

Modeling Language Definitions

Petri nets (PN). The first considered modeling language is the Petri net language, which served as running example in Chapter 5. A Petri net consists of places and transitions, which are connected for expressing input and output dependencies between transitions (cf. Figure 5.5 depicting the metamodel of the Petri net language).

The behavioral semantics of the Petri net language is specified in terms of tokens flowing through a Petri net (cf. Figure 5.6 and Figure 5.7). The initial marking of a Petri net, that is the initial distribution of tokens in the Petri net, is defined as input to the execution of the Petri net and new markings are induced by firing enabled transitions. Transitions are fired sequentially, the set of enabled transitions is recomputed after each firing, and enabled transitions are deterministically chosen for being fired.

Imperative modeling language (IML). The imperative modeling language allows the specification of computations with Integers. As defined by its metamodel depicted on the left-hand side of Figure 7.5, it provides modeling concepts for defining Integer variables possessing a name and an initial value (metaclass Variable), assignments through mathematical expressions over variables (metaclasses Assignment and Expression), and conditional goto statements (metaclasses Goto, Mark, and Condition).

The behavioral semantics of IML defines that the statements of an IML model are processed in their sequential order top to bottom and under the consideration of goto statements. Therefore, as can be seen on the right-hand side of Figure 7.5, we introduced a program counter into the xMOF-based configuration of IML holding the index of the next statement to be processed (attribute statementPointer of ModelConfiguration). The operation main() of the configuration class

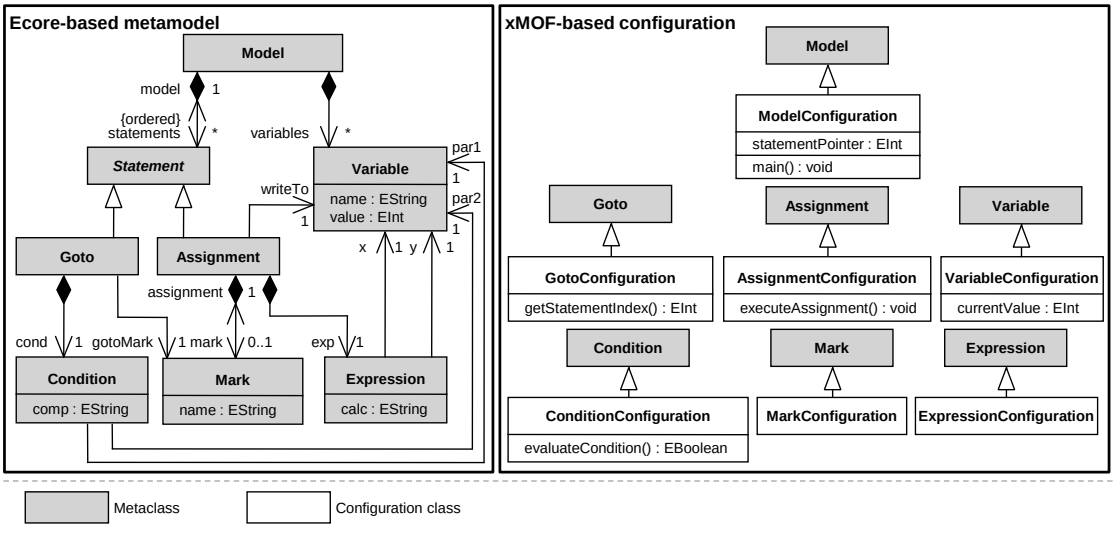


Figure 7.5: Imperative modeling language case study

ModelConfiguration is responsible for updating the program counter and triggering the execution of the next statement. The behavior of executing assignment statements and goto statements is defined by operations introduced into the configuration classes of the respective metaclass. The operation executeAssignment() of the configuration class AssignmentConfiguration defines how assignment statements are executed including the evaluation of expressions. The operation evaluateCondition() of the configuration class ConditionConfiguration defines how goto conditions are evaluated, and the operation getStatementIndex() of the configuration class GotoConfiguration defines how the target statement index of a goto statement is retrieved. For capturing the runtime values of variables, we introduced the attribute currentValue into the configuration class VariableConfiguration.

Finite state automata (FSA). The metamodel of the developed finite state automata language is depicted at the top of Figure 7.6. A finite state automata consists of a set of named states, where one state serves as initial state and several states may serve as accepting states (metaclass State). Transitions lead from one state to another state and are labeled with processable events (metaclass Transition).

The behavioral semantics of finite state automata is defined by the xMOF-based configuration depicted at the bottom of Figure 7.6 as described in the following. A finite state automata evaluates whether a sequence of input events is valid, i.e., if the sequential processing of the input events leads from the initial state to an accepting state. For processing one input event it is checked whether exactly one outgoing transition of the current state is labeled with the event. If this is the case, the current state is updated to the target state of this transition. Otherwise, the provided sequence of input events is invalid. The main() operation introduced into the configuration class FSAConfiguration takes as input the sequence of events to be processed (initialization

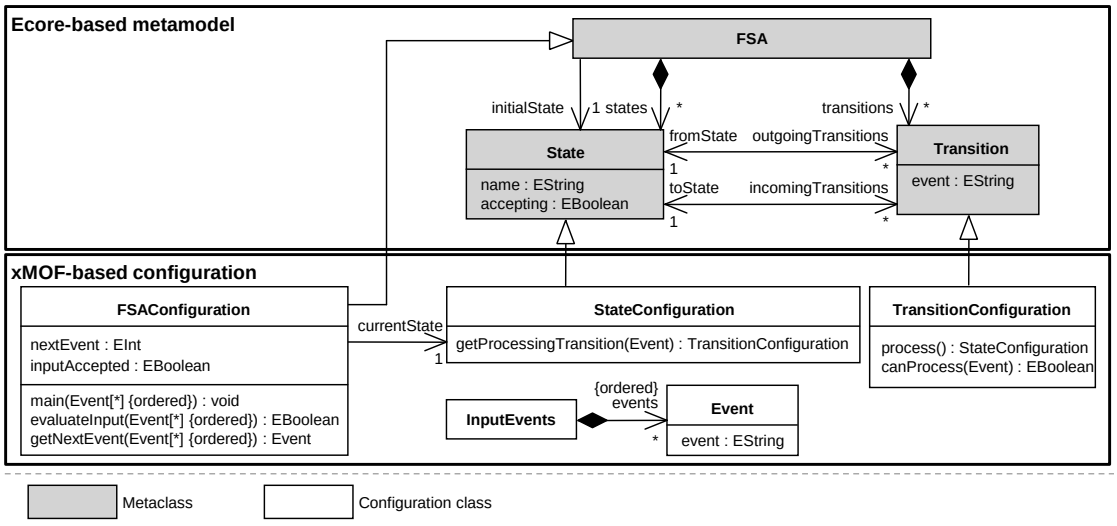


Figure 7.6: Finite state automata case study

classes InputEvents and Event). This sequence of events is evaluated by the operation evaluateInput(), which calls the operation getNextEvent() for retrieving the next event to be processed. For evaluating one event in the current state, first the transition labeled with this event is retrieved using the operation getProcessingTransition() of the configuration class StateConfiguration, which in turn uses the operation canProcess() of the configuration class TransitionConfiguration for determining whether a transition is labeled with this event. If such a transition is retrieved, the current state of the finite state automata is accordingly updated by calling the operation process() of the configuration class TransitionConfiguration. During the execution, the current state of a finite state automata, the next input event to be processed, as well as the final result of the execution are captured by the configuration class FSAConfiguration via the reference currentState, as well as the attributes nextEvent and inputAccepted.

UML activity diagrams (AD). The UML activity diagram case study is concerned with a variant of UML activity diagrams defined by Maoz *et al.* [95]. The metamodel, which we created for this case study depicted on the left-hand side of Figure 7.7, is based on the UML metamodel of the Eclipse UML 2 plugin and introduces additional modeling concepts that were considered by Maoz *et al.* In particular, the metamodel contains the following UML 2 standard conforming modeling concepts: activities, initial nodes, activity final nodes, fork nodes, join nodes, decision nodes, merge nodes, control flow edges, and opaque actions. Because Maoz *et al.* introduced

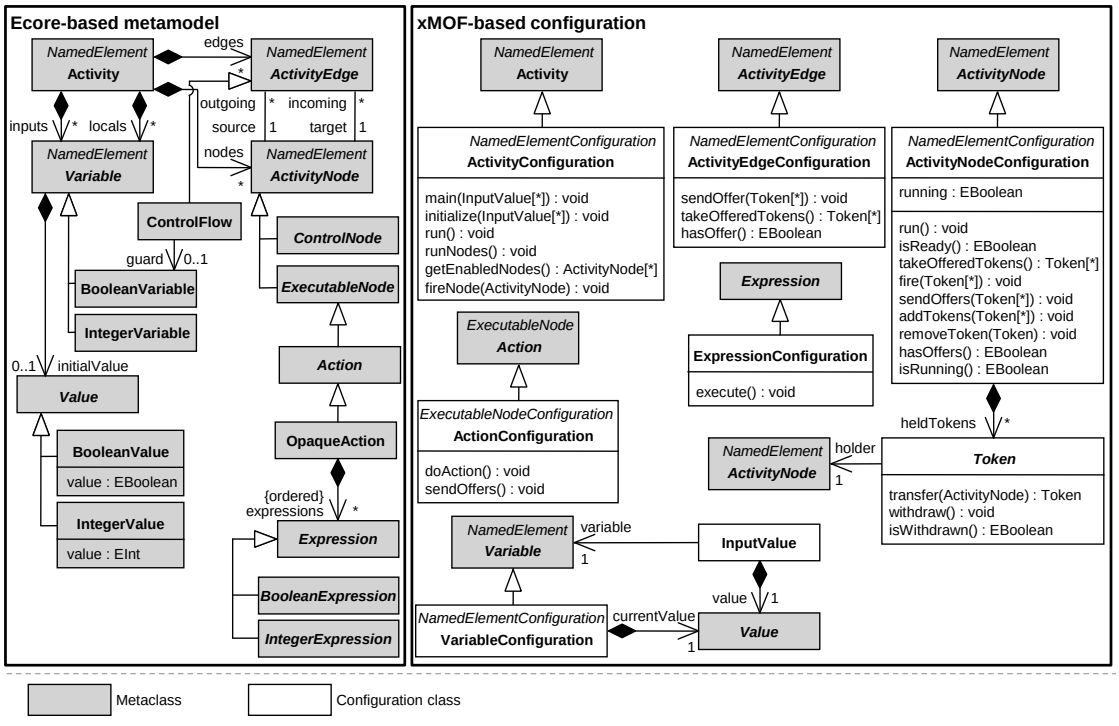


Figure 7.7: Activity diagram case study (excerpt)

local and input variables and allow the definition of expressions over these variables in actions as well as for guard conditions of control flow edges, we likewise introduced the following additional modeling concepts into our UML activity diagram metamodel: variables (metaclass Variable), values (metaclass Value), and expressions (metaclass Expression).

The behavioral semantics defined for the UML 2 compliant modeling concepts is based on the semantics defined by the fUML standard as depicted on the right-hand side of Figure 7.7. For instance, the operations doAction() and sendOffers() introduced into the configuration class ActionConfiguration correspond to the same named operations defined by the activation visitor class ActionActivation of the fUML execution model (cf. Figure 3.7). For executing expressions, we introduced the operation execute() into the configuration class ExpressionConfiguration and defined the behavior of this operation with activities contained by the concrete subclasses of ExpressionConfiguration. The runtime values of variables are captured by the containment reference currentValue of the configuration class VariableConfiguration. To enable the provision of input values for input variables of an activity (reference inputs of metaclass Activity), we added the initialization class InputValue to the xMOF-based configuration. The main() operation of the semantics specification owned by the configuration class ActivityConfiguration takes as input a set of InputValue instances.

UML class diagrams (CD). The last case study is concerned with UML class diagrams. For this case study, we extracted a subset of the UML metamodel of the Eclipse UML 2 plugin comprising modeling concepts for defining packages, classes, properties, associations, enumerations, and primitive types. An excerpt of the metamodel is depicted at the top of Figure 7.8.

An excerpt of the xMOF-based configuration, which we created for defining the behavioral semantics of UML class diagrams, is shown at the bottom of Figure 7.8. It consists of two parts. The first part is concerned with the representation of values, i.e., the runtime manifestation of instances of classifiers defined by a UML class diagram, such as instances of defined classes, associations, enumerations, and primitive types. Therefore, we introduced configuration classes into the xMOF-based configuration of UML class diagrams, which correspond to the semantic visitor classes defined by the fUML execution model for representing values

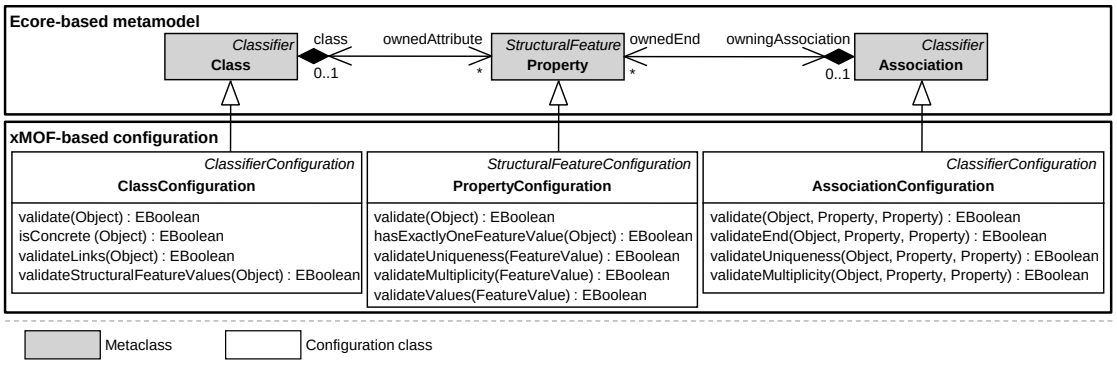


Figure 7.8: Class diagram case study (excerpt)

(cf. Figure 3.6). As an example, the configuration class Object defines the representation of class instances consisting of feature values holding an object's attribute values. Because the configuration classes for representing values are defined analogously to the semantic visitor classes defined by the fUML execution model for representing values, they are omitted in Figure 7.8. The second part of the xMOF-based configuration is concerned with evaluating whether a given object diagram is a valid instance of a defined class diagram. Therefore, we introduced validation operations into the configuration classes of the metaclasses Class, Property, Association, MultiplicityElement, PrimitiveType, Enumeration, and Type and defined their behavior with activities. Figure 7.8 shows the configuration classes ClassConfiguration, PropertyConfiguration, and AssociationConfiguration defining the behavioral semantics of the metaclasses Class, Property, and Association. For instance, the configuration class ClassConfiguration evaluates whether an object is a valid instance of the respective class by checking whether the class is a concrete class instead of an abstract class (operation isConcrete()) and whether the object's links and feature values comply with the associations and attributes of the class (operations validateLinks() and validateStructuralFeatureValues()). In the evaluation of object diagrams we considered concrete and abstract classes, inheritance relationships, class attributes of different types (primitive types, enumerations, classes), associations, and multiplicities as well as uniqueness constraints of class attributes and association ends.

Modeling Language Characteristics

The modeling languages considered in the case studies follow different *semantic paradigms*, namely (i) token flow semantics, (ii) state-based semantics, (iii) imperative programming, and (iv) declarative programming. The Petri net language and the considered variant of UML activity diagrams follow the semantic paradigm of token flows. Finite state automata have a state-based semantics as underlying semantics paradigm. The imperative modeling language and the modeling concepts for defining expressions provided by the considered UML activity diagram variant follow the imperative programming paradigm. Class diagrams constitute declarative definitions of valid classifier instantiations and, thus, follow the declarative programming paradigm.

The considered modeling languages also differ in their *complexity*. Table 7.2 shows size metric values of the Ecore-based metamodels as well as xMOF-based configurations of the modeling languages. Concerning the design size of the modeling language's metamodels measured by the number of contained metaclasses, the Petri net language, imperative modeling language, and finite state automata language are with three to eight metaclasses of rather small size. The design size of UML activity diagrams and UML class diagrams is with 31 and 37 metaclasses considerably larger. This difference in the complexity of the considered modeling language's metamodels is reflected by the developed behavioral semantics specifications. While the xMOF-based configurations of the Petri net language, imperative modeling language, and finite state automata language consist of five to seven classes, the xMOF-based configurations of UML activity diagrams and UML class diagrams comprise 38 and 56 classes, respectively. Also the complexity of the actual behavior definitions in terms of the number of activities defined in the xMOF-based configurations is higher for the UML activity diagram and UML class diagram case studies. They comprise 60 and 119 activities, respectively, compared to four to seven activities defined for the Petri net language, imperative modeling language, and finite state automata language.

Ecore-based metamodel	**PN**	**IML**	**FSA**	**AD**	**CD**
Metaclass	(3)	(8)	(3)	(31)	(37)
Non-abstract metaclass	3	7	3	19	25
Abstract metaclass	0	1	0	12	12
Attribute	2	5	3	8	35
Reference	(4)	(13)	(7)	(20)	(58)
Non-containment reference	2	13	6	14	55
Containment reference	2	0	1	6	3
Enumeration	0	0	0	4	3
Inheritance relationship	0	2	0	28	35

xMOF-based configuration	**PN**	**IML**	**FSA**	**AD**	**CD**
Class	(5)	(7)	(5)	(38)	(56)
Non-abstract class	5	7	5	37	50
Abstract class	0	0	0	1	6
Attribute	0	2	4	3	7
Reference	(3)	(0)	(3)	(11)	(14)
Non-containment reference	1	0	2	5	11
Containment reference	2	0	1	6	3
Inheritance relationship	3	7	3	33	52
Operation	7	4	6	43	88
Activity	7	4	6	60	119
Action	(38)	(54)	(54)	(312)	(742)
Add structural feature value action	3	5	8	22	2
Call behavior action	4	9	6	26	72
Call operation action	6	3	5	72	158
Clear structural feature action	0	0	0	1	0
Create object action	1	0	0	12	26
Destroy object action	1	0	0	1	0
Expansion region	5	1	2	18	58
Read extent action	0	0	0	0	1
Read is classified object action	0	1	0	1	30
Read self action	6	7	8	70	115
Read structural feature action	7	22	12	57	118
Remove structural feature value action	0	0	0	1	2
Structured activity node	0	0	0	2	0
Test identity action	0	2	4	4	58
Value specification action	5	4	9	25	102
Control node	(14)	(25)	(21)	(68)	(261)
Decision node	3	5	5	24	117
Fork node	8	16	13	34	132
Initial node	2	0	1	0	0
Merge node	1	4	2	10	12
Edge	(59)	(113)	(99)	(452)	(1,567)
Control flow	10	24	16	67	285
Object flow	49	89	83	385	1,282

Table 7.2: Size metric values of semantics specification case studies

7.2.3 Results

In the following, we discuss the conclusions drawn from performing the presented case studies with regard to answering the posed research questions.

Research Question 1: Adequacy of fUML

Expressiveness. The action language provided by fUML was expressive enough to completely define the behavioral semantics of the five considered executable modeling languages, such that we were able to successfully execute conforming models by means of fUML's execution environment. We only identified minor shortcomings of the foundational model library, which provides primitive behaviors for the primitive data types supported by fUML. For instance, we missed the list function indexOf, which we needed, e.g., to retrieve the index of the target statement of goto jumps in IML. However, it is easily possible to make additional primitive behaviors available at the fUML execution environment by providing and registering Java implementations of these primitive behaviors.

As can be seen in Table 7.2, we made use of 15 action types provided by fUML for defining the behavioral semantics of the considered modeling languages. From these 15 action types, call operation action, read structural feature action, read self action, value specification action, and call behavior action are the most often used ones accounting for 77% of actions contained by the developed semantics specifications. The action types clear structural feature action, read extent action, destroy object action, structured activity node, and remove structural feature value action were scarcely used. The remaining twelve action types provided by fUML were not required to define the behavioral semantics of the considered modeling languages. This includes action types related with active classes and asynchronous communication, link actions, and special types of structured activity nodes.

The high number of read structural feature actions led us to the observation that functions for navigating and querying models more concisely may ease the development of behavioral semantics specifications with fUML. Simple navigations over multiple associations require one action for each navigated association, which leads to long chains of reading actions. For collecting objects that fulfill certain conditions, even more complex combinations of reading actions, decision nodes, and merge nodes are required. Thus, the size and complexity of the fUML activities developed in our case studies could be reduced significantly, if a support for similar collection operations as provided by OCL, such as select and reject, would be provided.

The non-use of several action types can be explained by the fact that fUML contains modeling concepts that can be alternatively used for expressing the same behavior. For instance, a conditional node can be used alternatively to a set of decision nodes. Similarly, expansion regions and loop nodes can both be used to iterate over a set of values. For accessing and modifying associations, either structural feature actions or link actions can be used.

Suitability. fUML provides a computationally complete action language comprising actions for querying and manipulating objects, primitive behaviors for performing computations, as well as control structures for expressing complex control flows. Therewith, it is possible to define the behavior of UML classes in an object-oriented and imperative way. As UML classes

constitute the basis for MOF and metamodeling languages in general, we experienced fUML and particularly its action language as being directly adoptable for formally defining the behavioral semantics of executable modeling languages.

Concerning the standardized graphical UML activity diagram notation, which we used in the case studies to define the behavioral semantics of the considered modeling languages, we conclude that a combination with the textual concrete syntax defined by the Alf standard [117] might be more suitable than relying solely on the graphical notation. This is due to the observation, that activity diagrams easily become very large because it is required to model on a very detailed level. For instance, to increment an Integer attribute value of an object, at least five actions and four activity edges connecting them are required[4]. To mitigate readability issues, we aimed to keep the size of activities in terms of the number of contained activity nodes as small as possible in the behavioral semantics specifications developed in the course of the case studies. Nevertheless, for expressing computations, we find a textual notation to be more suitable. Thus, we conclude that while it might be useful to have a graphical view of activities that are not very detailed, i.e., which mainly call other activities, a textual representation might be more suitable for detailed activities implementing algorithms and computations. Therefore, we propose the combination of the graphical UML activity diagram notation and the textual concrete syntax defined by the Alf standard for defining the behavioral semantics of modeling languages with fUML.

Research Question 2: Adequacy of Methodology

Development of semantics specifications. In our case studies, we experienced that the processes, techniques, and supporting tools provided by the elaborated semantics specification methodology enable a systematic and efficient development of behavioral semantics specifications as they guided us through the semantics specification development.

The starting point for defining a modeling language's behavioral semantics is the modeling language's abstract syntax definition in the form of a metamodel. From the metamodel, an xMOF-based configuration is automatically initialized, which contains for each metaclass a configuration class that is subsequently augmented with the definition of the metaclasses's behavioral semantics. The *behavioral semantics of the metaclasses* is defined by introducing operations into the respective configuration classes and defining their behavior with fUML conforming activities in an object-oriented and imperative style. Due to the inheritance relationships between configuration classes and metaclasses, static information about the structure of a model element can be easily queried in the activities using reading actions provided by fUML.

Any *runtime concept* required for defining the behavioral semantics can be defined in the xMOF-based configuration by introducing attributes and references into the automatically initialized configuration classes or defining new configuration classes. For instance, in the Petri net language we defined a specific configuration class Token for capturing the markings of a Petri net. As the xMOF-based configuration is defined in its own artifact clearly separated from the

[4]One action each for (i) retrieving the object, (ii) retrieving the object's current attribute value, (iii) specifying the value one, (iv) calling the primitive behavior *plus*, (v) and updating the object's attribute value.

modeling language's metamodel, no modifications of the metamodel are required for defining additional runtime concepts.

Similar to runtime concepts, *inputs* required for executing a model can be defined in the xMOF-based configuration of a modeling language by introducing additional classes called initialization classes. Therewith, it is possible to define complex inputs consisting of arbitrarily many classes and references between them. In four of the five modeling languages developed in the case studies we defined such complex input parameters. The most complex input definition specified in the case studies is the input definition of the UML class diagram case study. It consists of 16 initialization classes, which enable the definition of complete object diagrams. The tool support of the semantics specification methodology developed for EMF enables these input definitions to be easily instantiated for defining concrete inputs for the execution of models.

Execution of models. The behavioral semantics specifications developed in the course of the case studies enabled us to directly execute models conforming to the considered modeling languages. Therefore, the semantics specification methodology defines its own processes and techniques, which are accordingly implemented by the supporting tools integrated with EMF. The execution of a model by means of the fUML execution environment requires the conversion of the model into an xMOF-based model being an instance of the defined xMOF-based configuration, a conversion of the xMOF-based configuration into an fUML model, and the conversion of the xMOF-based model into fUML extensional values. All these required steps are performed automatically and transparent to the user. The provided tool support requires the user only to define the xMOF-based configuration, the model to be executed, and the input definition.

Analysis of semantics specifications. The result of performing a model execution consists in the final runtime state of the executed model, which allows reasoning about the result of the model execution carried out. For instance, in the Petri net language, the final runtime state of an executed Petri net comprises the final distribution of tokens among the Petri net's places and enables reasoning about the Petri net's final marking. To more thoroughly analyze a performed model execution, the event mechanism, command interface, and trace model introduced into the fUML execution environment can be utilized. We observed that the runtime information provided by trace models is adequate for verifying the correctness of semantics specifications developed with xMOF. The trace models captured during performing model executions enabled us to thoroughly test the behavioral semantics specifications developed in the course of the case studies. For instance, for the UML class diagram case study, we created 41 JUnit tests for verifying whether the behavioral semantics specification is correct, that is, whether example object diagrams are correctly evaluated by the execution of a UML class diagram. Because the model execution environment provided for xMOF builds directly on top of the extended fUML execution environment, any analysis method and technique implemented for fUML on top of the extended fUML execution environment, can be straightforwardly integrated with xMOF to provide the same analysis methods and techniques for analyzing behavioral semantics specifications with xMOF. Thus, the debugger and testing framework for fUML developed on top of the extended fUML execution environment as presented in Section 7.1.2 could be easily in-

tegrated with xMOF to provide a debugger and testing framework for xMOF-based semantics specifications.

Research Question 3: Non-Invasiveness of fUML Integration

Language integration. From applying the proposed language integration strategy for integrating fUML with Ecore, we conclude that this strategy is non-invasive. The integration did not require any modifications of Ecore's meta-metamodel and, consequently, any techniques and tools built upon Ecore's meta-metamodel remained unaffected by the integration.

Semantics specification methodology. The developed semantics specification methodology considers the metamodel of a modeling language as being the starting point for defining the modeling language's behavioral semantics. Thus, it does not interfere with any existing metamodeling methodologies or metamodeling environments but only provides concrete processes, techniques, and supporting tools for defining a modeling language's behavioral semantics with fUML.

Semantics specification. When applying the proposed semantics specification methodology, the definition of a modeling language's behavioral semantics is clearly separated from the modeling language's abstract syntax definition in its own artifact. Thereby, the behavioral semantics definition extends the abstract syntax definition of the modeling language. Technically speaking, this is achieved by the introduction of inheritance relationships between metaclasses defining the abstract syntax of modeling concepts and configuration classes defining the behavioral semantics of modeling concepts. Due to this clear separation, the metamodel of a modeling language is not affected by the semantics specification. As a consequence, any tools support derived from or built upon the metamodel of a modeling language, such as editors, model transformations, and code generators, are not affected by the behavioral semantics specification.

7.3 Semantic Model Differencing

7.3.1 Research Questions

In contrast to existing semantic model differencing approaches, our approach follows the spirit of generic syntactic model differencing by utilizing the behavioral semantics specification of a modeling language to perform semantic model differencing. Therefore, we proposed a generic semantic model differencing framework that performs semantic differencing by executing the models to be compared for relevant inputs, capturing execution traces adhering to a generic execution trace format, and syntactically comparing the captured execution traces for identifying semantic differences among the models. Our approach is generally applicable for realizing semantic model differencing operators for any modeling language whose behavioral semantics is defined using an operational semantics approach allowing the execution of models and capture of execution traces complying to our format. Furthermore, it is configurable with respect to the semantic equivalence criterion to be applied in semantic model differencing through the definition of suitable semantic match rules.

We evaluated our generic semantic model differencing framework concerning its expressive power provided for defining realistic semantic model differencing operators as well as concerning the performance of semantic model differencing operators developed with our framework. In particular, we aimed at answering the following research questions.

Research question 1: Expressive power of generic semantic model differencing. *Is our generic semantic model differencing framework expressive enough to develop non-trivial semantic model differencing operators?*

Research question 2: Performance of generic semantic model differencing. *How performant are semantic model differencing operators developed with our generic semantic model differencing framework?*

We instantiated our generic semantic model differencing framework for the fUML-based behavioral semantics specification approach presented in Chapter 5. Therefore, we implemented an Eclipse plugin responsible for carrying out the generic functionalities of our framework, namely syntactic matching of models based on syntactic match rules defined with ECL, invoking the model execution environment for obtaining execution traces, and semantic matching of models based on semantic match rules defined with ECL, captured execution traces, and established syntactic model correspondences. Furthermore, we implemented on top of our model execution environment for fUML-based behavioral semantics specification an Eclipse plugin responsible for capturing execution traces adhering to our generic execution trace format. Therefore, we utilized the event mechanism provided by our extended fUML execution environment presented in Chapter 4.

Using this instantiation of our framework, we carried out two case studies in which we developed semantic model differencing operators for distinct modeling languages and applied them to example models [86]. By drawing conclusions from the development of the semantic model differencing operators we answered the first research question regarding the expressive power of our generic semantic model differencing framework. By applying the developed semantic model differencing operators to example models and measuring the execution time needed for performing the semantic differencing, we answered the research question regarding the performance of semantic model differencing operators developed with our framework.

7.3.2 Case Studies

To evaluate whether our generic semantic model differencing framework provides the expressive power necessary to define non-trivial semantic model differencing operators, we carried out two case studies, in which we implemented semantic model differencing operators for UML activity diagrams and UML class diagrams. To ensure that the semantic model differencing operators developed in the course of the case studies are realistic and, thus, non-trivial, we implemented them according to the semantic model differencing operators ADDiff [95] and CDDiff [96] developed by Maoz *et al.* This allowed us to evaluate whether our generic framework is powerful enough to develop semantic model differencing operators comparable to already existing differ-

encing operators specifically design for a particular modeling language. Therefore, we defined the behavioral semantics of UML activity diagrams and UML class diagrams according to the definitions provided by Maoz *et al.* using our fUML-based behavioral semantics specification approach. We already presented these behavioral semantics specifications of UML activity diagrams and UML class diagrams in Section 7.2. In the following, we briefly recap our definitions of the abstract syntax and behavioral semantics of UML activity diagrams and UML class diagrams, and discuss the semantic equivalence criteria implemented by the developed semantic model differencing operators.

UML activity diagrams. The UML activity diagram variant considered in this case study is defined in accordance with the definitions of Maoz *et al.* [95]. It provides the following modeling concepts: activities, control nodes, opaque actions, control flow edges, local and input variables, as well as expressions. The behavioral semantics is based on the fUML standard and corresponds to the semantics defined by Maoz *et al.*

ADDiff applies the semantic equivalence criterion *trace equivalence* for UML activity diagrams. Two activities diagrams are semantically equivalent if all sequences of action executions possible in one activity diagram are also possible in the other activity diagram. Sequences of action executions that are only possible in one of two activity diagrams constitute diff witnesses. We defined semantic match rules corresponding to this semantic equivalence criterion.

UML class diagrams. The UML class diagram variant considered in this case study provides according to the definitions of Maoz *et al.* [96] the following class modeling concepts: packages, classes, properties, associations, enumerations, and primitive types.

Similarly to the UML activity diagram case study, we aimed at adopting the semantics definition of UML class diagrams provided by Maoz *et al.* Maoz *et al.* defined the semantics of UML class diagrams using a translational approach. A class diagram is translated into an Alloy module consisting of predicates that specify the semantics of the class diagram. By utilizing the Alloy Analyzer and, in particular, the embedded SAT solver SAT4J, object diagrams conforming to these predicates and, hence, conforming to the class diagram are generated. However, as implementing our own efficient SAT solving algorithms with fUML was out of scope of this evaluation, we decided to define a different semantics for UML class diagrams, which, nevertheless, implements the same semantics of classifier instantiation considered by Maoz *et al.* Instead of generating objects diagrams conforming to a given class diagram, we defined the behavioral semantics of UML class diagrams in terms of evaluating whether a given object diagram conforms to the class diagram. Thereby, we considered concrete and abstract classes, inheritance relationships, class attributes of different types (primitive types, enumerations, classes), associations, as well as multiplicities and uniqueness constraints of class attributes and association ends.

In the semantic model differencing of two UML class diagrams, CDDiff tries—again using the Alloy Analyzer—to generate object diagrams that conform to one class diagram but not to the other one and, hence, constitute diff witnesses. Due to the decision to define for this case study a different semantics for UML class diagrams than applied by CDDiff, we also had to adapt the semantic equivalence criterion applied for semantic model differencing. Our semantic

model differencing operator does not generate object diagrams conforming to only one of two compared class diagram, but instead determines whether a set of given input object diagrams contains object diagrams which conform to only one of two compared class diagrams. Therefore, our semantic match rules retrieve the evaluation results obtained from the execution of both class diagrams for the same object diagram and determines whether they are equal. If this is the case, the object diagram either conforms to both class diagrams or to none and, hence, the class diagrams are semantically equivalent for this object diagram. Otherwise, if the evaluation results of the compared class diagrams for the same object diagram are not equal, the object diagram conforms to only one the two compared class diagram and, hence, the class diagrams are not semantically equivalent.

7.3.3 Setup

For verifying the correctness of the semantic model differencing operators developed in the case studies as well as for evaluating their performance, we applied them to the same example models Maoz *et al.* used for evaluating ADDiff and CDDiff[5].

The inputs to be considered in the semantic model differencing of the example models had to be provided manually because an implementation of symbolic model execution for generating the inputs is not yet integrated with our prototype. Thus, in the UML activity diagram case study, we had to manually define input values for global variables defined by the example activity diagrams. In the UML class diagram case study, more complex inputs were required, namely complete object diagrams.

To verify the correctness of our developed semantic model differencing operators, we applied them to the example models with inputs for which both compared model versions behave the same with respect to the applied semantic equivalence criterion as well as with inputs for which they behave differently.

The performance evaluation was done by measuring the execution time needed for performing the semantic model differencing of selected example models. We measured the execution time needed for performing the syntactic matching of the compared models, executing the compared models for a given set of inputs, and performing the semantic matching based on the execution traces obtained from the model execution. The execution time was measured by taking timestamps right before the start and right after the completion of each of these three steps. Additionally, we measured the execution time needed for performing each single model execution, i.e., for executing each model on one input, by taking timestamps right before the start and right after the completion of each model execution. In the same manner, we measured the execution time needed for performing each single semantic matching, i.e., the application of the semantic match rules on two execution traces. This performance evaluation was performed on the following hardware and software environment.

- **Hardware:** Intel Dual Core i5-2520M CPU 2.5 GHz, 8 GB RAM
- **Operating system:** Windows 7 Professional, Service Pack 1

[5] `http://www.se-rwth.de/materials/semdiff`, accessed 05.09.2014

- **Eclipse:** Kepler, Service Release 1, Build 20130919-0819
- **Java:** Version 7, Update 3, Build 1.7.0_03-b05

7.3.4 Results

In the following, we discuss our conclusions drawn from the case studies with regard to answering the posed research questions.

Research Question 1: Expressive Power of Generic Semantic Model Differencing

Our semantic model differencing operators developed for UML activity diagrams and UML class diagrams detected the same diff witnesses among the used example models as ADDiff and CDDiff. In the case of UML activity diagrams, both our semantic model differencing operator as well as ADDiff reported the same sequences of action executions as diff witnesses. In the case of UML class diagrams, our semantic model differencing operator correctly reported semantic differences for object diagrams which where generated by CDDiff as diff witnesses.

Due to these results, our overall conclusion drawn from performing the case studies is, that our generic semantic model differencing framework provides *sufficient expressive power* for defining non-trivial semantic model differencing operators. Additionally to this overall conclusion, we made the following observations about the development of semantic model differencing operators with our generic framework.

Developing semantic model differencing operation is a language engineering task. For developing a semantic model differencing operator with our generic framework, semantic match rules have to be implemented that compare execution traces according to a suitable semantic equivalence criterion. Implementing these semantic match rules requires besides knowledge about model comparison languages, such as ECL, thorough knowledge about the behavioral semantics specification of the considered modeling language. In particular, knowledge about the structure of a model's runtime state, which is captured in the execution traces to be compared, is needed, as well as knowledge about how the model's runtime state is manipulated during its execution. Both is defined by the behavioral semantics specification of the considered modeling language. Thus, we regard the development of semantic model differencing operators as a language engineering task.

Runtime concepts build the basis for semantic model differencing. Our generic semantic model differencing framework realizes semantic model differencing as syntactic comparison of execution traces. Thus, all information about a model's runtime behavior that is required for performing semantic differencing has to be captured by execution traces. For example, in the UML activity diagram case study, information about the execution of actions is required for realizing a semantic model differencing operator applying the trace equivalence criterion.

Our generic execution trace format captures the runtime behavior of a model in terms of sequences of the model's runtime states during its execution. The runtime state of a model is expressed using the runtime concepts defined by the behavioral semantics specification of the used

modeling language. Thus, any runtime information needed in the semantic model differencing has to be defined in terms of runtime concepts by the modeling language's behavioral semantics specification.

Research Question 2: Performance of Generic Semantic Model Differencing

We measured the performance of the developed semantic model differencing operators for UML activity diagrams and UML class diagrams in terms of the execution time needed for semantically differencing selected example models.

Table 7.3 shows the measured execution times distinguishing between syntactic matching (SynMatching), model execution (Execution), semantic matching (SemMatching), and total time needed (Total). Besides the measured execution times, also some size metrics about the example models are shown, which influence the measured values. For the UML activity diagram examples, the number of contained activity nodes (#Nodes) is shown, which influences the performance of the model execution step. Furthermore, the number of defined input variables (#Variables) is shown, which influences the number of inputs (#Inputs) to be considered in the semantic model differencing and, therewith, the performance of the model execution step as well as the semantic matching step. For the UML class diagram examples, the number of objects (#Objects) and the number of links (#Links) contained by the input object diagrams are shown, which both influence the performance of the model execution step. For comparing two class diagrams, only one input object diagram (#Inputs) was used in this evaluation.

The performance results indicate that the model execution is the most expensive step in the semantic model differencing as it takes around 95% of the overall execution time. Thus, the main reason for the weaker performance of our semantic model differencing operators compared to ADDiff and CDDiff is the performance of the model execution carried out using the extended fUML execution environment, which is based on the fUML virtual machine. However, as discussed in the performance evaluation of the extended fUML execution environment presented in Section 7.1.3, the capturing of execution traces compromises the performance of the fUML virtual machine significantly. We expect that a more efficient implementation of the trace capturing capability will lead to a significant improvement of the performance of semantic

UML activity diagram

Example	#Nodes	#Variables	#Inputs	SynMatching	Execution	SemMatching	Total
Anon V1/V2	15/15	2/2	4	51 ms	7905 ms	341 ms	8297 ms
Anon V2/V3	15/19	2/3	8	72 ms	7374 ms	246 ms	7692 ms
hire V1/V2	14/15	1/1	2	47 ms	5259 ms	283 ms	5589 ms
hire V2/V3	15/15	1/1	2	47 ms	5745 ms	304 ms	6096 ms
hire V3/V4	15/15	1/1	2	48 ms	2011 ms	95 ms	2154 ms
IBM2557 V1/V2	18/16	1/1	6	85 ms	25889 ms	1159 ms	27133 ms

UML class diagram

Example	#Objects	#Links	#Inputs	SynMatching	Execution	SemMatching	Total
EMT V1/V2	2	1	1	17 ms	1203 ms	119 ms	1339 ms
EMT V1/V2	4	3	1	16 ms	6790 ms	275 ms	7081 ms
EMT V1/V2	6	5	1	15 ms	26438 ms	543 ms	26996 ms

Table 7.3: Execution time measurements for semantic model differencing case studies

model differencing operators developed with our generic semantic model differencing framework. Furthermore, with an implementation of symbolic execution for fUML as introduced in Section 6.5, the model execution step might become obsolete, as it might be possible to perform semantic model differencing based on execution trees and symbolic states computed through the symbolic execution of the models to be compared.

CHAPTER 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we presented contributions towards automating the development of semantics-based tools for executable modeling languages by providing a language for formally defining the behavioral semantics of modeling languages and a generic model execution environment building the common basis for a variety of semantics-based tools. In the following, we summarize the contributions elaborated in the course of this thesis as well as conclusions drawn from their evaluation.

Contribution 1: Semantics specification with fUML. The lack of a widely accepted or even standardized language for formally defining the behavioral semantics of modeling languages hampers the emergence of techniques for automating the development of semantics-based tools, such as tools for model debugging, model testing, and dynamic model analysis. In this thesis, we proposed fUML to become this behavioral semantics specification language, because it is like the metamodeling language MOF standardized by the OMG and based on UML, its behavioral semantics is formally defined, and an execution environment for fUML models is available. We elaborated a *language integration strategy* for integrating fUML into existing metamodeling languages and applied this strategy for integrating fUML with Ecore. The integration results in an executable metamodeling language that enables the formal definition of the behavioral semantics of modeling languages using fUML's activity modeling concepts. Moreover, we proposed a *semantics specification methodology*, including dedicated tool support for EMF, for developing executable modeling languages with fUML fostering a clear separation of concerns and enabling the execution of models using fUML's execution environment. Based on case studies, we conclude that fUML is a very promising candidate for being established as a standardized language for defining the behavioral semantics of modeling languages. With this contribution we aim at providing a stimulus towards the adoption of fUML as common semantics specification language in MDE.

Contribution 2: Extensions of the fUML execution environment. Semantics-based tools relying on the capability to execute models, such as tools for model debugging, model testing, and dynamic model analysis, depend heavily on an execution environment providing means for runtime observation, execution control, and runtime analysis. To enable the development of semantics-based tools for fUML models, we consequently incorporated such means into fUML's execution environment. In particular, we developed an *event mechanism* that enables the observation of the state of model executions during runtime, a *command interface* that provides control over model executions enabling their suspension and resumption, as well as a *trace model* that captures runtime information about carried out model executions sufficient for conducting runtime analyses. We incorporated these extensions into the reference implementation of fUML's execution environment. Based on the resulting extended fUML execution environment, we were able to develop a debugger, a testing framework, as well as a non-functional property analysis tool for fUML models. The development of these tools led us to the conclusion that our extensions of the fUML execution environment are adequate for implementing debugging, testing, dynamic analysis, and non-functional property analysis tools.

Contribution 3: Semantics-based tool development. Based on fUML's execution environment and our extensions of this environment, we developed a *generic model execution environment* for executable modeling languages defined with fUML. This generic model execution environment enables the execution of models based on the formal definition of the used modeling language's behavioral semantics specified with fUML. Furthermore, it provides means for runtime observation, execution control, and runtime analysis. Therewith, the generic model execution environment builds a common basis for developing semantics-based tools for executable modeling languages. This includes in particular tools for model testing, model debugging, and dynamic model analysis. We have shown how this is possible by developing a *model execution tool* as well as a *model debugger* on top of the generic model execution environment. Like the generic model execution environment itself, also these tools are implemented in a generic manner and, hence, can be directly utilized for executing and debugging models that conform to any executable modeling language defined with fUML. With the generic model execution environment for executable modeling languages defined with fUML, we aim at laying the basis for future innovations regarding the automated development of semantics-based tools.

Contribution 4: Semantic model differencing. Existing semantic model differencing approaches suggest the translation of models into a semantic domain suitable for expressing the models' semantics and calculating semantic differences, as well as the implementation of analysis algorithms dedicated to semantic differencing in this semantic domain. In contrast, we developed a *generic semantic model differencing framework* that enables the development semantic differencing operators based on the behavioral semantics specifications of modeling languages. The generic framework extracts semantic interpretations of models in terms of execution traces from the behavioral semantics specification by executing the models using the execution environment provided by the employed semantics specification approach. By comparing the extracted semantic interpretations syntactically, semantic differences among the models with respect to a suitable semantic equivalence criterion may be identified. Thus, non-trivial transfor-

mations into a semantic domain specifically for enabling semantic differencing can be avoided. We developed an instantiation of this framework for our fUML-based semantics specification approach presented as contribution 1, which utilizes the generic model execution environment developed as part of contribution 3 and relying on contribution 2 for capturing execution traces. Using this instantiation, we evaluated our generic model differencing approach resulting in the conclusion that it provides sufficient expressive power to define non-trivial semantic model differencing operators.

8.2 Future Work

In the following, we discuss interesting directions for future work building upon the research conducted in the course of this thesis. These future research directions originate from our experiences gained during the implementation of prototypes as well as from the evaluation results obtained by applying the prototypes in case studies. They concern on the one hand identified limitations of solutions proposed in this thesis and on the other hand research topics continuing our efforts towards automating the development of semantics-based tools for executable modeling languages.

Performance improvements of fUML execution environment. Our approach for automating the development of semantics-based tools consists in the provision of a generic model execution environment that operates on fUML-based behavioral semantics specifications and provides means for runtime observation, execution control, and runtime analysis. As a consequence, the performance of the generic model execution environment substantially influences the performance of any semantics-based tool built on top of it. Thereby, the performance of the generic model execution environment itself basically equates to the performance of fUML's execution environment as well as our introduced extensions.

The performance evaluation of our extensions of the fUML execution environment resulted into the observation that the way these extensions are implemented—that is, using aspect-orientated programming—seriously compromises the performance of fUML's execution environment. Furthermore, during the conduct of our evaluations, we also experienced scalability issues of fUML's execution environment itself especially in terms of memory consumption. However, it has to be noted that our implementation relies on the reference implementation of fUML's execution environment, whose primarily purpose is to serve as reference for tool vendors, not to provide a high-performance implementation.

To mitigate these performance issues, we recommend the design and implementation of an execution environment for fUML that on the one hand directly integrates means for runtime observation, execution control, and runtime analysis as proposed by our event mechanism, command interface, and trace model, and on the other hand aims at high performance and scalability.

Integrated development environment for fUML-based behavioral semantics specifications. Semantics-based tools developed using our approach operate on fUML-based behavioral semantics specifications. Thus, ensuring the correctness of fUML-based behavioral semantics specifications is crucial in order to ensure the correctness and usefulness of semantics-based

tools building upon them. Therefore, methods and techniques supporting the development of high-quality behavioral semantics specifications with fUML are required.

We envision an integrated development environment for fUML-based behavioral semantics specifications that support language designers in constructing high-quality semantics specifications with fUML. Functionalities provided by such an integrated development environment include on the one hand efficient editing support, such as editors with high usability and refactoring support, and on the other hand tools for performing analyses, such as debugging, testing, dynamic analysis, and formal analysis. Some of these methods and techniques have been already developed for fUML. For instance, in this thesis, we have introduced a debugger and a testing framework for fUML models that may be easily adopted for fUML-based behavioral semantics specifications. Other examples are the debugging techniques for fUML elaborated by Laurent *et al.* [88] and the lightweight verification methods for Alf developed by Planas *et al.* [130].

Reuse, specialization, and integration of behavioral semantics specifications. Another interesting line of future work is concerned with providing means for reusing, specializing, and integrating behavioral semantics specifications. The need for such means may be illustrated on the example of UML. Several of our case studies are concerned with dedicated members of the UML language family, in particular, with UML class diagrams and UML activity diagrams. If we consider the remainder of UML, we can clearly identify the need for reusing, specializing, and integrating behavioral semantics specifications. For instance, UML includes several modeling concepts having the same or similar behavioral semantics. An example are accept event actions to be used in UML activity diagrams and transitions to be used in UML state machines both having the behavior of receiving events. Besides such reuse potentials, UML comprises a plethora of semantic variation points allowing the specialization of the behavioral semantics of dedicated UML modeling concepts. Furthermore, the various kinds of behaviors in UML, namely, activities, state machines, and interactions, may be connected with each other requiring to also connect their behavioral semantics specifications in general. The illustrated need for reusing, specializing, and integrating behavioral semantics specification is, however, not specific to UML. For instance, the behavioral semantics of the diverse types of workflow modeling languages, such as UML activity diagrams, BPMN, and Petri nets, are based on token flow semantics. Thus, dedicated means for reusing behavioral semantics specifications might increase the efficiency of developing behavioral semantics specifications. Furthermore, current research efforts are directed towards providing support for integrating heterogeneous modeling languages in order to enable their coordinated use in the development of software systems requiring knowledge from diverse domains [4, 29]. In this context, the integration of behavioral semantics specifications is an important research topic.

Automated development of semantics-based tools. With our approach of formally defining the behavioral semantics of executable modeling languages with fUML, we aim at laying the basis for the emergence of techniques enabling the automated development of semantics-based tools. In the course of this thesis, we developed a generic model execution environment for executable modeling languages defined with fUML. This generic model execution environment can be used to efficiently develop semantics-based tools relying on the capability to observe, control,

and analyze model executions. Based on the examples of a model execution tool, a model debugger, and a semantic model differencing framework, we showed how semantics-based tools can be developed in a generic manner based on the generic model execution environment. However, the development of further generic semantics-based tools operating on fUML-based behavioral semantics specifications, such as tools implementing testing techniques and dynamic analysis techniques, is left for future work. Furthermore, we have not considered the automated development of tools implementing formal analysis methods in this thesis.

Symbolic execution for fUML. Symbolic execution is a technique for exploring all possible execution paths through a program and generating inputs that lead to the execution of these explored paths. We have proposed to apply symbolic execution to fUML in order to generate inputs for models that lead to the execution of all possible paths of the used modeling language's fUML-based behavioral semantics specification. These inputs induce all possible distinct execution traces of a model and may, hence, be used to perform semantic model differencing following our approach. Besides this application domain, symbolic execution for fUML may be also applied to generate test inputs for fUML models as well as for fUML-based behavioral semantics specifications. These test inputs might not only be used for generating tests suites providing high test coverage, but also for debugging, as inputs may be generated that lead to the execution of a particular statement or that lead to a particular system state. We regard symbolic execution as a potential technique for developing high-quality fUML models and high-quality fUML-based behavioral semantics specifications. Thus, developing a symbolic execution environment for fUML is definitely an interesting topic for future research.

Development of further executable modeling languages with fUML. In the course of this thesis, we have developed several behavioral semantics specifications for executable modeling languages using our fUML-based semantics specification approach. However, in future work we would like to use our approach for defining the behavioral semantics of further executable modeling languages in order to demonstrate its usefulness and gather additional requirements on development methods for behavioral semantics specifications. It would be particularly interesting to develop executable modeling languages having diverse application domains and diverse behavioral semantics including, for instance, model query languages, model transformation languages, and further sub-languages of UML. The latter would also constitute an additional contribution itself towards providing a complete and formal definition of the behavioral semantics of UML.

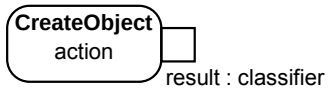
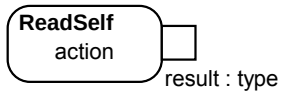
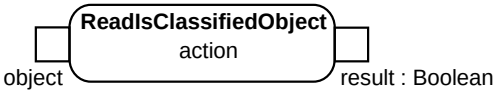
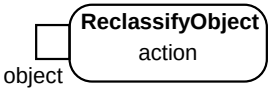
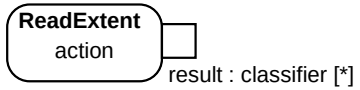
APPENDIX A

fUML Action Language

In Chapter 3, we provided a thorough introduction to the fUML standard giving an overview of the UML subset considered by fUML, as well as the fUML virtual machine capable of executing fUML models. The main modeling concept for describing the behavior of a system with fUML is the modeling concept Activity. An activity consists of activity nodes connected through activity edges (cf. Figure 3.4). We distinguish between three kinds of activity nodes, namely object nodes, control nodes, and actions. Object nodes can be further divided into input and output pins of actions, as well as activity parameter nodes of activities. Control nodes available in fUML are initial node, activity final node, fork node, join node, decision node, and merge node. Furthermore, the fUML subset contains 27 types of actions defined by UML's action language, which can be categorized into object actions, structural feature actions, link actions, communication actions, and structured activity nodes.

In the following tables, we provide a detailed description of the action types (Tables A.1, A.2, A.3, A.4, A.5, A.6) as well as control node types (Table A.7) contained by the fUML subset. Each table consists of three columns showing (i) the action or control node in graphical concrete syntax (optional elements are shown in gray color), (ii) important attributes and references that have to be set, and (iii) a short textual description of the behavior of the action or control node.

Further details on the modeling concepts contained by the fUML subset can be found in the fUML standard [114] as well as in the UML standard [111].

Action	Structural features	Description
	classifier : Classifier	Creates an instance of the defined classifier.
	isDestroyLinks : Boolean isDestroyOwnedObjects : Boolean	Destroys the provided object. isDestroyLinks = true : links of the provided object are destroyed isDestroyOwnedObjects = true : objects owned by the provided object are destroyed
	-	Provides the context object of the activity execution. In case the activity defines the behavior of an operation, the object for which the operation was called is provided. Otherwise, the activity execution itself is provided.
	classifier : Classifier isDirect : Boolean	Determines whether the provided object is an instance of the defined classifier. isDirect = true : direct instantiation is determined
	oldClassifier : Classifier [*] newClassifier : Classifier [*] isReplaceAll : Boolean	Removes the defined old classifiers from the provided object's types and adds the defined new classifiers as types. isReplaceAll = true : all classifiers not defined as new classifiers are removed
	classifier : Classifier	Retrieves all instances of the defined classifier currently existing at the locus of the execution.

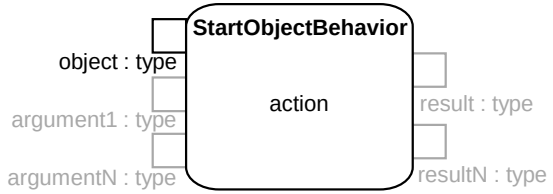
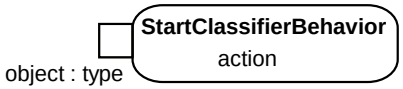
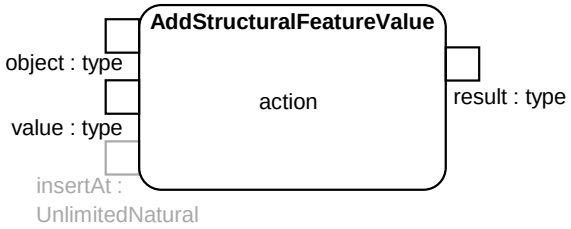
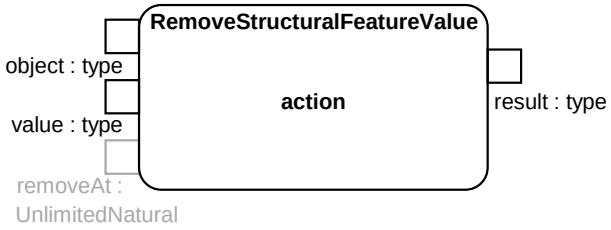
Action	Structural features	Description
	-	Starts the behavior of the provided object. If the provided object is an instance of a behavior, this behavior is executed. Otherwise, the classifier behavior of the provided object's type is started.
	-	Starts the classifier behavior of the provided object.

Table A.1: fUML object actions

Action	Structural features	Description
	structuralFeature : StructuralFeature isReplaceAll : Boolean	Adds the provided value to the provided object for the defined structural feature at the provided position. isReplaceAll = true : existing values for the defined structural feature of the provided object are removed before the provided value is added
	structuralFeature : StructuralFeature isRemoveDuplicates : Boolean	Removes the provided value for the defined structural feature from the provided object. isRemoveDuplicates = true : duplicates of the provided value for the defined structural feature are removed from the provided object

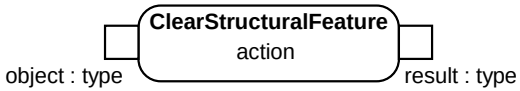
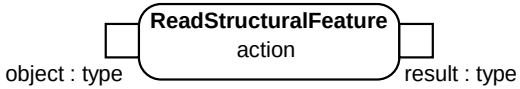
Action	Structural features	Description
	structuralFeature : StructuralFeature	Removes all values for the defined structural feature from the provided object.
	structuralFeature : StructuralFeature	Reads the values for the defined structural feature of the provided object.

Table A.2: fUML structural feature actions

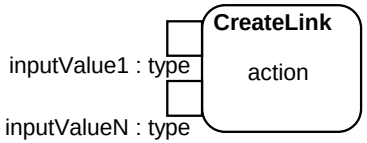
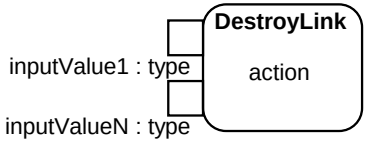
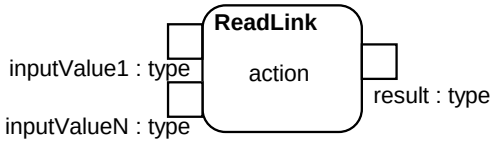
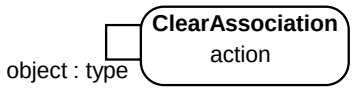
Action	Structural features	Description
	endData : LinkEndCreationData [2..*]	Creates a link between the provided input objects according to the defined end data.
	endData : LinkEndDestructionData [2..*]	Destroys the links existing between the provided input objects according to the defined end data.
	endData : LinkEndData [2..*]	Reads the objects linked with the provided objects according to the defined end data by navigating the respective association towards one end.
	association : Association	Destroys all links of the provided object for the defined association.

Table A.3: fUML link actions

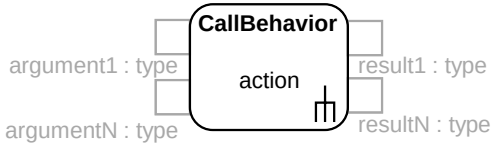
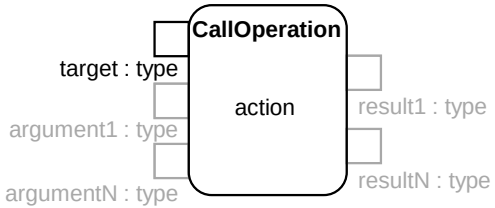
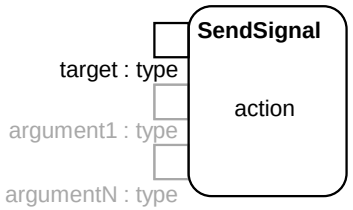
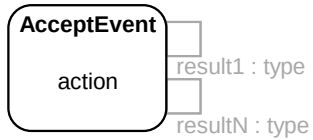
Action	Structural features	Description
	behavior : Behavior	Invokes the defined behavior synchronously.
	operation : Operation	Invokes the defined operation on the provided target object synchronously.
	signal : Signal	Sends a signal according to the defined signal type to the provided target object asynchronously.
	trigger : Trigger [1..*]	Accepts a signal event occurrence according to the defined trigger.

Table A.4: fUML communication actions

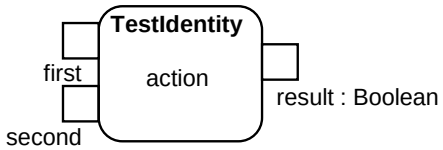
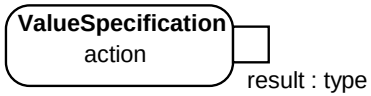
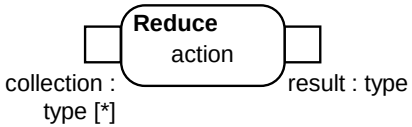
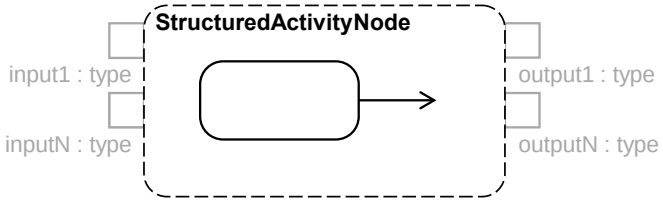
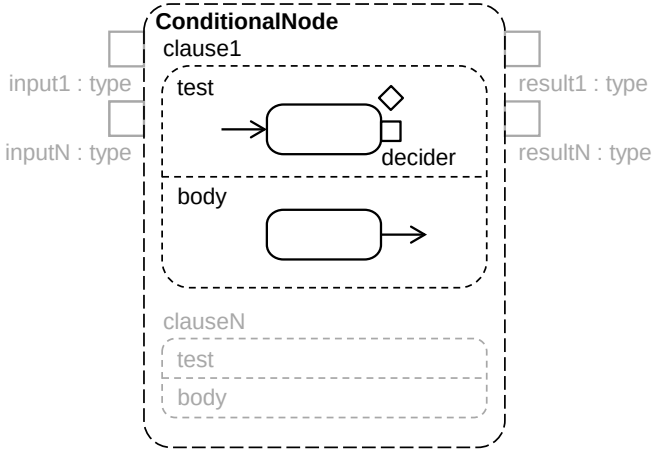
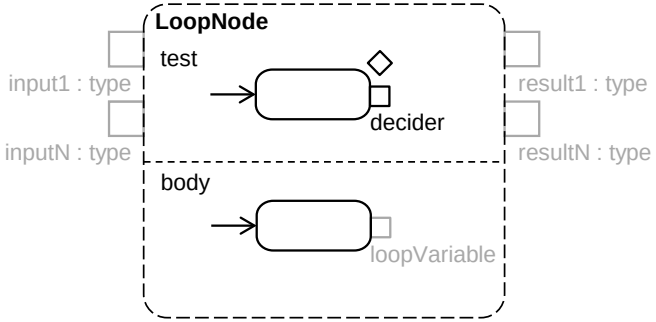
Action	Structural features	Description
	-	Tests whether two values are identical.
	value : ValueSpecification	Provides a value according to the defined specification.
	reducer : Behavior isOrdered : Boolean	Reduces the provided collection of values to a single value by applying the defined reducer behavior. isOrdered = true : the provided values are processed by the reducer behavior in the order defined by the collection

Table A.5: fUML other actions

Action	Structural features	Description
	-	Executes the contained activity nodes.

Action	Structural features	Description
	-	Executes the body section of exactly one clause whose test section evaluates to true.
	isTestFirst : Boolean	Executes the body section as long as the test section evaluates to true. isTestFirst = true : test section is executed always before the body section is executed

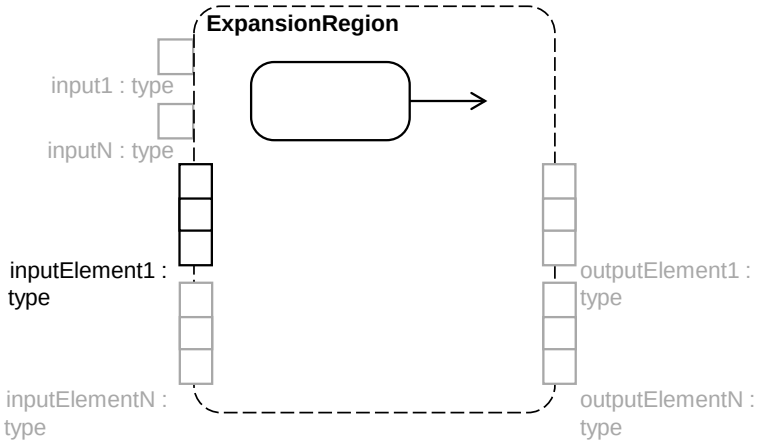
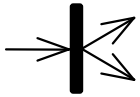
Action	Structural features	Description
	mode : ExpansionKind	Executes the contained activity nodes for each collection element provided through the input element expansion nodes. mode = iterative : collection elements are processed iteratively mode = parallel : collection elements are processed in parallel

Table A.6: fUML structured activity nodes

Control node	Structural features	Description
 initial node	-	Starts the control flow of an activity.
 activity final node	-	Stops all flows of an activity.
 fork node	-	Splits a flow into multiple concurrent flows.

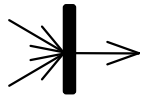
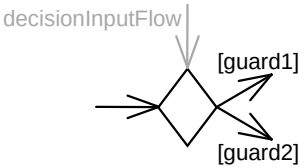
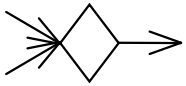
Control node	Structural features	Description
 join node	-	Synchronizes multiple concurrent flows.
 decision node	decisionInput : Behavior [0..1]	Decides between multiple alternative flows by calculating a decision value and comparing it with the guard specifications defined for the outgoing activity edges. Decision value calculation: (i) decisionInputFlow = null and decisionInput = null : decision value is equal to the value provided via the incoming object flow (ii) decisionInputFlow != null and decisionInput = null : decision value is equal to the value provided via the decision input flow (iii) decisionInputFlow != null and decisionInput != null : decision value is calculated by executing the defined decision input behavior on the value provided via the decision input flow and the value provided by the incoming object flow, if any
 merge node	-	Merges multiple alternative flows.

Table A.7: fUML control nodes

APPENDIX B

Implementations

The languages, concepts, methods, techniques, and tools elaborated in the course of this thesis have been implemented as research prototypes. The prototypes are openly available under the Eclipse Public License Version 1.0[1] and can be found in our public source code repository[2]. Table B.1 provides an overview of the implemented prototypes including information about the source code projects realizing the prototypes as well as references to the sections of this thesis discussing the prototypes. If not stated otherwise, the source code projects are available in the *"default"* repository[3]. Further information about the prototypes may also be found at our project website[4].

[1] `http://www.eclipse.org/legal/epl-v10.html`, accessed 11.09.2014

[2] `http://code.google.com/a/eclipselabs.org/p/moliz`, accessed 11.09.2014

[3] `http://code.google.com/a/eclipselabs.org/p/moliz/source/checkout?repo=default`, accessed 11.09.2014

[4] `http://www.modelexecution.org`, accesses 11.09.2014

Prototype	Component	Project	Reference
fUML execution environment extensions	Extensions (event mechanism, command interface, trace model)	org.modelexecution.fumldebug.core org.modelexecution.fumldebug.core.aspect	4.2, 4.3, 4.4
	Extended execution environment	org.modelexecution.fumldebug	4
	Performance evaluation	org.modelexecution.fumldebug.eval org.modelexecution.fumldebug.eval.extensions org.modelexecution.fumldebug.eval.noextensions	7.1.3
	Debugger	org.modelexecution.fuml.convert.uml2 org.modelexecution.fumldebug.debugger org.modelexecution.fumldebug.debugger.papyrus org.modelexecution.fumldebug.debugger.ui org.modelexecution.fumldebug.standardlibrary	7.1.2
	Testing framework	Available in *"testing"* repository[5]	
	Non-functional property analysis tool	Available in *"nfr"* repository[6]	
Semantics specification language	Language xMOF (fUML integrated with Ecore)	org.modelexecution.xmof	5.2.2
	Primitive behavior library	org.modelexecution.xmof.standardlibrary	3.2.3
	Evaluation	org.modelexecution.xmof.examples	7.2.2
Semantics specification methodology	Wizard for initializing xMOF-based configurations	org.modelexecution.xmof.configuration	5.3
	Editor for xMOF-based configurations	org.modelexecution.xmof.diagram org.modelexecution.xmof.edit org.modelexecution.xmof.editor	
	Wizards for instantiating initialization classes and defining inputs	org.modelexecution.xmof.ui	
Generic model execution environment	Model execution environment	org.modelexecution.xmof.vm	5.4
	xMOF to fUML converter	org.modelexecution.fuml.convert.xmof	

[5] `http://code.google.com/a/eclipselabs.org/p/moliz/source/checkout?repo=testing`, accessed 11.09.2014
[6] `http://code.google.com/a/eclipselabs.org/p/moliz/source/checkout?repo=nfr`, accessed 11.09.2014

Prototype	Component	Project	Reference
Generic model execution tool and debugger	Runtime profile and runtime profile application generator[7]	org.modelexecution.xmof.configuration.profile	5.5.1
	Launch configuration for model execution and debugging	org.modelexecution.xmof.debug org.modelexecution.xmof.debug.ui	5.5.1, 5.5.2
	Debugger configuration	org.modelexecution.xmof.modeldebugger org.modelexecution.xmof.modeldebugger.edit org.modelexecution.xmof.modeldebugger.editor	5.5.2
Semantic model differencing	Generic semantic model differencing framework	org.modelexecution.xmof.diff	6.2, 6.3, 6.4
	Generic execution trace format and execution trace capturing for fUML	org.modelexecution.xmof.states	6.4.1, 6.4.2
	Evaluation	org.modelexecution.xmof.diff.test	7.3

Table B.1: Overview of research prototypes implemented in the course of this thesis

[7] For defining runtime profiles and runtime profile applications, we make use of EMF Profiles available at `http://code.google.com/a/eclipselabs.org/p/emf-profiles`.

List of Figures

List of Tables

Listings

Bibliography

[1] Islam Abdelhalim, Steve Schneider, and Helen Treharne. Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP. In *Proceedings of the 13th International Conference on Formal Engineering Methods and Software Engineering (ICFEM'11)*, volume 6991 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2011.

[2] 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.

[3] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *Proceedings of the 6th International Conference on the Unified Modeling Language: Modeling Languages and Applications (UML'03)*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[4] Vasco Amaral, Cécile Hardebolle, Hans Vangheluwe, László Lengyel, and Peter Bunus. Summary of the Workshop on Multi-Paradigm Modelling: Concepts and Tools. In *Models in Software Engineering: Workshops and Symposia at MODELS 2011, Reports and Revised Selected Papers*, volume 7167 of *Lecture Notes in Computer Science*, pages 83–88. Springer, 2012.

[5] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[6] Thomas Ball. The Concept of Dynamic Analysis. *SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[7] Simonetta Balsamo, Antinisca di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[8] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2011.

[9] Abderraouf Benyahia, Arnaud Cuccuru, Safouan Taha, François Terrier, Frédéric Boulanger, and Sébastien Gérard. Extending the Standard Execution Model of UML for Real-Time Systems. In *Proceedings of the 7th IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES'10) and the 3rd IFIP TC 10 International Conference on Biologically Inspired Collaborative Computing (BICC'10)*, volume 329 of *IFIP Advances in Information and Communication Technology*. Springer, 2010.

[10] Luca Berardinelli, Philip Langer, and Tanja Mayerhofer. Combining fUML and Profiles for Non-functional Analysis Based on Model Execution Traces. In *Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA'13)*, pages 79–88. ACM, 2013.

[11] Simona Bernardi, José Merseguer, and Dorina C. Petriu. Dependability Modeling and Analysis of Software Systems Specified with UML. *ACM Computing Surveys*, 45(1):2:1–2:48, 2012.

[12] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[13] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 273–280. IEEE, 2001.

[14] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[15] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM Semantics for UML Activity Diagrams. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2000.

[16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[17] Erwan Breton and Jean Bézivin. Towards an Understanding of Model Executability. In *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS'01)*, pages 70–80. ACM, 2001.

[18] Manfred Broy and María Victoria Cengarle. UML formal semantics: lessons learned. *Software & Systems Modeling*, 10(4):441–446, 2011.

[19] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[20] Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai. Challenges and Directions in Formalizing the Semantics of Modeling Languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.

[21] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pages 239–254. ACM, 2014.

[22] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[23] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic Anchoring with Model Transformations. In *Proceedings of the 1st European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05)*, volume 3748 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2005.

[24] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. Ceteva, 2nd edition, 2008.

[25] Lori A. Clarke. A Program Testing System. In *Proceedings of the 1976 Annual Conference (ACM'76)*, pages 488–491. ACM, 1976.

[26] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[27] Adrian Colyer, Andy Clement, George Harley, and Mathew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley, 2005.

[28] 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.

[29] Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *IEEE Computer*, 47(6):68–71, 2014.

[30] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[31] Michelle L. Crane and Jürgen Dingel. On the Semantics of UML State Machines: Categorization and Comparison, 2005. Technical Report 2005-501. School of Computing, Queen's University. `http://ftp.qucis.queensu.ca/TechReports/Reports/2005-501.pdf`.

[32] Michelle L. Crane and Jürgen Dingel. Towards a Formal Account of a Foundational Subset for Executable UML Models. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*, volume 5301 of *Lecture Notes in Computer Science*, pages 675–689. Springer, 2008.

[33] Michelle L. Crane and Jürgen 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 (CASCON'08)*, pages 8:96–8:110. ACM, 2008.

[34] Juan de Lara and Hans Vangheluwe. Using AToM^3 as a Meta-Case Tool. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS'02)*, pages 642–649, 2002. `http://www.cs.mcgill.ca/~hv/publications/02.ICEIS.MCASE.pdf`.

[35] Thomas Degueule, Olivier Barais, Arnaud Blouin, and Benoît Combemale. The K3 Model-Based Language Workbench, 2014. Technical Report hal-01025283, Version 1. L'Institut National de Recherche en Informatique et e n Automatique (INRIA), Université de Rennes, Institut National des Sciences Appliquées (INSA). `http://hal.archives-ouvertes.fr/hal-01025283`.

[36] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech) co-loaced with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 31–36. ACM, 2007.

[37] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A Systematic Approach to Generate Inputs to Test UML Design Models. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 95–104. IEEE Computer Society, 2006.

[38] Trung T. Dinh-Trong, Sudipto Ghosh, Robert B. France, Michael Hamilton, and Brent Wilkins. UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In *Proceedings of the 2005 Workshop on Eclipse Technology eXchange (ETX) co-located with the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 120–124. ACM, 2005.

[39] Trung T. Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert B. France, and Anneliese Andrews. A Tool-Supported Approach to Testing UML Design Models. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 519–528. IEEE Computer Society, 2005.

[40] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-specific

Languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12)*, pages 112–121. ACM, 2012.

[41] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard (UML'00)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[42] Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In *Proceedings of the 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.

[43] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.

[44] Rik Eshuis and Roel Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.

[45] Rik Eshuis and Roel Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[46] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets, 1994. Technical Report BRICS RS-94-8, BRICS Report Series. Department of Computer Science, University of Aarhus. `http://www.brics.dk`.

[47] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wasowski. Sound Merging and Differencing for Class Diagrams. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE'14)*, volume 8411 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2014.

[48] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. Vision Paper: Make a Difference! (Semantically). In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, volume 6981 of *Lecture Notes in Computer Science*, pages 490–500. Springer, 2011.

[49] Martin Fleck, Luca Berardinelli, Philip Langer, Tanja Mayerhofer, and Vittorio Cortellessa. Resource Contention Analysis of Service-Based Systems through fUML-Driven Model Execution. In *Proceedings of the 5th International Workshop on Non-functional Properties in Modeling: Analysis, Languages and Processes (NiM-ALP) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, volume 1074, pages 6–15. CEUR, 2013.

[50] Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[51] Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. Execution and Simulation of (Profiled) UML Models using Pópulo. In *Proceedings of the International Workshop on Modeling in Software Engineering (MiSE) co-located with the 30th International Conference on Software Engineering (ICSE'08)*, pages 75–81, 2008.

[52] Alessandro Gerlinger Romero, Klaus Schneider, and Maurício Gonçalves Vieira Ferreira. Using the Base Semantics given by fUML for Verification. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14)*, pages 5–16. SCITEPRESS Digital Library, 2014.

[53] Christian Gerth, Jochen M. Küster, Markus Luckey, and Gregor Engels. Precise Detection of Conflicting Change Operations Using Process Model Terms. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, volume 6395 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2010.

[54] Christian Gerth, Markus Luckey, Jochen M. Küster, and Gregor Engels. Detection of Semantically Equivalent Fragments for Business Process Model Change Management. In *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC'10)*, pages 57–64. IEEE, 2010.

[55] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398, 2005.

[56] 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.

[57] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 16–27. ACM, 2003.

[58] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, volume 6394 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2010.

[59] Lars Hamann, Oliver Hofrichter, and Martin Gogolla. On Integrating Structure and Behavior Modeling with OCL. In *Proceedings of the the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12)*, volume 7590 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2012.

[60] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'04)*, pages 42–55. IBM Press, 2004.

[61] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A metamodel for the compact but lossless exchange of execution traces. *Software & Systems Modeling*, 11(1):77–98, 2012.

[62] David Harel. Biting the Silver Bullet - Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, 1992.

[63] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[64] Jan Hendrik Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2005. `http://digital.ub.uni-paderborn.de/hs/id/3928`.

[65] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating OCL and Textual Modelling Languages. In *Models in Software Engineering: Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2011.

[66] Brian Henderson-Sellers. UML - the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software & Systems Modeling*, 4(1):4–13, 2005.

[67] Alan R. Hevner. The Three Cycle View of Design Science. *Scandinavian Journal of Information Systems*, 19(2):87–92, 2007.

[68] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[69] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.

[70] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480. ACM, 2011.

[71] International Standards Organization. ISO 18629, Process Specification Language, 2004. `http://www.nist.gov/psl`.

[72] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 201–221. Springer, 2011.

[73] Paul C. Jorgensen. *Modeling Software Behavior: A Craftsman's Approach*. Auerbach Publications, 2009.

[74] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2006.

[75] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society, 2008.

[76] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[77] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2008.

[78] Dimitrios Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2009.

[79] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, and Richard Paige. *The Epsilon Book*. March 2014. `https://www.eclipse.org/epsilon/doc/book`.

[80] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS'11)*, volume 6705 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2011.

[81] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[82] Qinan Lai and Andy Carpenter. Defining and Verifying Behaviour of Domain Specific Language with fUML. In *Proceedings of the 4th Workshop on Behaviour Modelling - Foundations and Applications (BM-FA) co-located with the 8th European Conference on Modelling Foundations and Applications (ECMFA'12)*, pages 1:1–1:7. ACM, 2012.

[83] Qinan Lai and Andy Carpenter. Static Analysis and Testing of Executable DSL Specification. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD'13)*, pages 157–162. SCITEPRESS Digital Library, 2013.

[84] Danny B. Lange and Yuichi Nakamura. Object-Oriented Program Tracing and Visualization. *IEEE Computer*, 30(5):63–70, 1997.

[85] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A Generic Framework for Realizing Semantic Model Differencing Operators. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14)*, volume 1258, pages 16–20. CEUR, 2014.

[86] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14)*, volume 8767 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2014.

[87] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012.

[88] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Executing and Debugging UML Models: an fUML extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, pages 1095–1102. ACM, 2013.

[89] Johan Lilius and Iván Porres Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of the 2nd International Conference on the Unified Modeling Language: Beyond the Standard (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–444. Springer, 1999.

[90] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4):349–361, 2007.

[91] Lockheed Martin Corporation, Model Driven Solutions. Foundational UML Reference Implementation. `http://portal.modeldriven.org/content/fuml-reference-implementation-download`.

[92] Shahar Maoz. Model-Based Traces. In *Models in Software Engineering: Workshops and Symposia at MODELS 2012, Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 109–119. Springer, 2009.

[93] Shahar Maoz. Using Model-Based Traces as Runtime Models. *IEEE Computer*, 42(10):28–36, 2009.

[94] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Models in Software Engineering: Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 2011.

[95] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proceedings of the 13th European Software Engineering Conference and the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pages 179–189. ACM, 2011.

[96] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, volume 6813 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2011.

[97] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A Runtime Model for fUML. In *Proceedings of the 7th Workshop on Models@run.time (MRT) co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12)*, pages 53–58. ACM, 2012.

[98] Tanja Mayerhofer, Philip Langer, and Manuel Wimmer. Towards xMOF: Executable DSMLs Based on fUML. In *Proceedings of the 12th Workshop on Domain-Specific Modeling (DSM) co-located with the 3rd Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'12)*, pages 1–6. ACM, 2012.

[99] Tanja Mayerhofer, Philip Langer, and Manuel Wimmer. xMOF: A Semantics Specification Language for Metamodeling. In *Joint Proceedings of Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, volume 1115, pages 46–50. CEUR, 2013.

[100] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs Based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*, volume 8225 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2013.

[101] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[102] Stephen J. Mellor, Stephen Tockey, Rodolphe Arthaud, and Philippe Leblanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In *Proceedings of the 1st International Workshop on the Unified Modeling Language: Beyond the Notation (UML'98)*, volume 1618 of *Lecture Notes in Computer Science*, pages 307–318. Springer, 1999.

[103] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[104] 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 (MoDeVVa) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, volume 1069, pages 1–10. CEUR, 2013.

[105] 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 (MinoPro) and 1st International Workshop on Event Modeling and Processing in Business Process Management (EMoV) co-located with Modellierung 2014*, volume 1185, pages 73–85. CEUR, 2014.

[106] Model Driven Solutions. Executable UML/SysML Semantics Project Report (Final), November 2008. `http://lib.modeldriven.org/MDLibrary/trunk/Applications/fUML-Reference-Implementation/trunk/doc/xUML-SysML-Project-Report.pdf`.

[107] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[108] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., 1992.

[109] Object Management Group. Action Semantics for the UML, Request For Proposal, September 1999. `http://www.omg.org/cgi-bin/doc?ad/98-11-01.pdf`.

[110] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification, Version 1.1, January 2005. `http://www.omg.org/spec/SPTP/1.1`.

[111] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3, May 2010. `http://www.omg.org/spec/UML/2.3`.

[112] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011. `http://www.omg.org/spec/QVT/1.1`.

[113] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011. `http://www.omg.org/spec/UML/2.4.1`.

[114] 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`.

[115] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, June 2011. `http://www.omg.org/spec/MARTE/1.1`.

[116] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.3, June 2012. http://www.omg.org/spec/SysML/1.3.

[117] Object Management Group. Action Language for Foundational UML (Alf), Version 1.0.1, October 2013. http://www.omg.org/spec/ALF/1.0.1.

[118] Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5, September 2013. http://www.omg.org/spec/UML/2.5/Beta2.

[119] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1, August 2013. http://www.omg.org/spec/FUML/1.1.

[120] Object Management Group. Object Constraint Language, Version 2.4, February 2014. http://www.omg.org/spec/OCL/2.4.

[121] Object Management Group. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.2, April 2014. http://www.omg.org/spec/MOF/2.4.2.

[122] Object Management Group. Precise Semantics of UML Composite Structures (PSCS), Version Beta 1, June 2014. http://www.omg.org/spec/PSCS/1.0/Beta1.

[123] Dirk Ohst, Michael Welle, and Udo Kelter. Differences Between Versions of UML Diagrams. *SIGSOFT Software Engineering Notes*, 28(5):227–236, 2003.

[124] Greg O'Keefe. Improving the Definition of UML. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, volume 4199 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2006.

[125] Hamilton Oliveira, Leonardo Murta, and Cláudia Werner. Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In *Proceedings of the 12th International Workshop on Software Configuration Management (SCM) co-located with the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 1–16. ACM, 2005.

[126] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'08)*, pages 226–237. ACM, 2008.

[127] Dorin B. Petriu and Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software & Systems Modeling*, 6(2):163–184, 2007.

[128] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert B. France. Testing UML designs. *Information & Software Technology*, 49(8):892–912, 2007.

[129] Orest Pilskalns, Gunay Uyan, and Anneliese Andrews. Regression Testing UML Designs. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 254–264, 2006.

[130] Elena Planas, Jordi Cabot, and Cristina Gómez. Lightweight Verification of Executable Models. In *Proceedings of the 30th International Conference Conceptual Modeling (ER'11)*, volume 6998 of *Lecture Notes in Computer Science*, pages 467–475. Springer, 2011.

[131] Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. Models in Conflict - Detection of Semantic Conflicts in Model-based Development. In *Proceedings of the 3rd International Workshop on Model-Driven Enterprise Information Systems (MDEIS) co-located with the 9th International Conference on Enterprise Information Systems (ICEIS'07)*, pages 29–40, 2007.

[132] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *Proceedings of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.

[133] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *Proceedings of the 1st International Conference on Software Language Engineering (SLE'08)*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2009.

[134] José Eduardo Rivera and Antonio Vallecillo. Adding Behavioral Semantics to Models. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07)*, pages 169–180. IEEE Computer Society, 2007.

[135] Daniel A. Sadilek and Guido Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'08)*, volume 5095 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2008.

[136] Markus Scheidgen and Joachim Fischer. Human Comprehensible and Machine Processable Specifications of Operational Semantics. In *Proceedings of the 3rd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, volume 4530 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2007.

[137] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[138] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[139] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[140] Bran Selic. The Less Well Known UML. In *Formal Methods for MDE*, volume 7320 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012.

[141] Koushik Sen. Concolic Testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 571–572. ACM, 2007.

[142] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[143] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988.

[144] Connie U. Smith, Catalina M. Lladó, and Ramon Puigjaner. Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. *Performance Evaluation*, 67(7):548–568, 2010.

[145] Christian Soltenborn. *Quality Assurance with Dynamic Meta Modeling*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2013. `http://digital.ub.uni-paderborn.de/hs/id/887158`.

[146] Christian Soltenborn and Gregor Engels. Towards Test-Driven Semantics Specification. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, volume 5795 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2009.

[147] Christian Soltenborn and Gregor Engels. Towards Generalizing Visual Process Patterns. In *Proceedings of the Workshop on Visual Formalisms for Patterns co-located with the 2009 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC'09)*, 2010. `http://journal.ub.tu-berlin.de/eceasst/article/view/345/332`.

[148] Ian Sommerville. *Software Engineering*. Pearson Education, 8th edition, Addison-Wesley.

[149] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling - State of the Art and Research Challenges. In *Revised Selected Papers of the International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2007.

[150] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2008.

[151] Harald Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC'04)*, pages 235–242. IEEE, 2004.

[152] Harald Störrle. Semantics of Exceptions in UML 2.0 Activities, 2004. Technical Report. Ludwig-Maximilians-Universität München. http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/V/AD3-Exceptions-TR.pdf.

[153] Harald Störrle. Semantics of Structured Nodes in UML 2.0 Activities. In *Proceedings of the 2nd Nordic Workshop on UML, Modeling, Methods and Tools (NWUML'04)*, 2004. http://www.pst.informatik.uni-muenchen.de/~stoerrle/V/AD4-Expansion.pdf.

[154] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52, 2005.

[155] Harald Störrle and Jan Hendrik Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In *Proceedings of Software Engineering*, volume 64 of *Lecture Notes in Informatics*, pages 117–128. GI, 2005.

[156] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software Practice and Experience*, 39(15):1253–1292, 2009.

[157] Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Principles for the Realization of an Open Simulation Framework Based on fUML (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium (DEVS'13)*, pages 4:1–4:6. Society for Computer Simulation International, 2013.

[158] Juha-Pekka Tolvanen, Matti Rossi, and Jeff Gray. Guest editorial to the theme issue on domain-specific modeling in theory and applications. *Software & Systems Modeling*, 13(1):5–7, 2014.

[159] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[160] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[161] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[162] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[163] 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, 2008.

[164] Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by Unified Model Analysis (PUMA). In *Proceedings of the 5th International Workshop on Software and Performance (WOSP'05)*, pages 1–12. ACM, 2005.

[165] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM, 2005.

[166] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 3rd edition, 2006.

Curriculum Vitae

Dipl.-Ing. Tanja Mayerhofer, BSc

Richtergasse 1a/5
1070 Wien
Austria

Email: mayerhofer@big.tuwien.ac.at
Web: http://www.big.tuwien.ac.at/staff/tmayerhofer
Date of Birth: 19-Aug-1987
Nationality: Austria

Education

2011 - 2014	**PhD Studies Business Informatics** Vienna University of Technology, Austria Supervision: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
2009 - 2011	**Master Studies Business Informatics** Vienna University of Technology, Austria Emphasis on Project and Quality Management
2006 - 2009	**Bachelor Studies Business Informatics** Vienna University of Technology, Austria Emphasis on Practical Software Engineering

Work Experience *(Excerpt)*

07/2011 - present	**Researcher** Business Informatics Group, Vienna University of Technology, Austria Research Interests: Model Driven Engineering Teaching: Model Engineering, Advanced Model Engineering, Web Engineering, Practicals, Bachelor's Theses, Master's Theses
03/2011 - 07/2011	**Tutor** Business Informatics Group, Vienna University of Technology, Austria Teaching: Web Engineering
11/2010 - 02/2011	**Teaching Assistant** Business Informatics Group, Vienna University of Technology, Austria Teaching: Evaluation and Implementation of the E-Learning System for the Course on Object-Oriented Modeling
09/2010 - 01/2011	**Tutor** SBA Research Teaching: Internet Security

Awards

2013	**Third Place, ACM Student Research Competition, Graduate Category** 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13) Submission Title: *Using fUML as Semantics Specification Language in Model Driven Engineering*
2011	**Diploma Thesis Award of the City Vienna** Thesis Title: *Breathing New Life into Models: An Interpreter-Based Approach for Executing UML Models*

Publications

Peer Reviewed Conference Papers

Philip Langer, Tanja Mayerhofer, Gerti Kappel. **Semantic Model Differencing Utilizing Behavioral Semantics Specifications**. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14),* volume 8767 of *Lecture Notes in Computer Science*, pages 116-132, Springer, 2014.

Philip Langer, Tanja Mayerhofer, Manuel Wimmer, Gerti Kappel. **On the Usage of UML: Initial Results of Analyzing Open UML Models**. In *Proceedings of Modellierung 2014,* vol-

ume 225 of *Lecture Notes in Informatics*, pages 289-304, GI, 2014.

Tanja Mayerhofer, Philip Langer, Manuel Wimmer, Gerti Kappel. **xMOF: Executable DSMLs based on fUML**. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13),* volume 8225 of *Lecture Notes in Computer Science*, pages 56-75, Springer, 2013.

Luca Berardinelli, Philip Langer, Tanja Mayerhofer. **Combining fUML and Profiles for Non-Functional Analysis Based on Model Execution Traces**. In *Proceedings of the 9th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA'13),* pages 79-88, ACM, 2013.

Peer Reviewed Workshop Papers, Demonstrations, and Poster Presentations

Philip Langer, Tanja Mayerhofer, Gerti Kappel. **A Generic Framework for Realizing Semantic Model Differencing Operators**. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14),* volume 1258, pages 16-20, CEUR, 2014.

Patrick Neubauer, Tanja Mayerhofer, Gerti Kappel. **Towards Integrating Modeling and Programming Languages: The Case of UML and Java**. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages (GEMOC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14),* volume 1236, pages 23-32, CEUR, 2014.

Stefan Mijatov, 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 (MinoPro) and 1st International Workshop on Event Modeling and Processing in Business Process Management (EMoV) co-located with Modellierung 2014,* volume 1185, pages 73-85, CEUR, 2014.

Tanja Mayerhofer. **Using fUML as Semantics Specification Language in Model Driven Engineering**. In *Joint Proceedings of Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13),* volume 1115, pages 87-93, CEUR, 2013.

Tanja Mayerhofer, Philip Langer, Manuel Wimmer. **xMOF: A Semantics Specification Language for Metamodeling**. In *Joint Proceedings of Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13),* volume 1115, pages 46-50, CEUR, 2013.

Stefan Mijatov, Philip Langer, Tanja Mayerhofer, 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 (MoDeVVa) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13),* volume 1069, pages 1-10, CEUR, 2013.

Martin Fleck, Luca Berardinelli, Philip Langer, Tanja Mayerhofer, Vittorio Cortellessa. **Resource Contention Analysis of Service-Based Systems through fUML-Driven Model Execution**. In *Proceedings of the 5th International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes (NIM-ALP) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13),* volume 1074, pages 6-15, CEUR, 2013.

Tanja Mayerhofer, Philip Langer, Gerti Kappel. **A Runtime Model for fUML**. In *Proceedings of the 7th International Workshop on Models@run.time (MRT) co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12),* pages 53-58, ACM, 2012.

Tanja Mayerhofer, Philip Langer, Manuel Wimmer. **Towards xMOF: Executable DSMLs Based on fUML**. In *Proceedings of the 12th Workshop on Domain-Specific Modeling (DSM) co-located with the 3rd Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'12),* pages 1-6, ACM, 2012.

Tanja Mayerhofer, Philip Langer. **Moliz: A Model Execution Framework for UML Models**. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12),* pages 3:1-3:2, ACM, 2012.

Tanja Mayerhofer. **Testing and Debugging UML Models Based on fUML**. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12),* Doctoral Symposium, pages 1579-1582, IEEE, 2012.

Marion Brandsteidl, Tanja Mayerhofer, Martina Seidl, Christian Huemer. **Replacing Traditional Classroom Lectures with Lecture Videos - An Experience Report**. In *Proceedings of the 8th Educators' Symposium: Software Modeling in Education (EduSymp) co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12),* pages 21-27, ACM, 2012.