FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Code Generation for fUML

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Ulrich Fischer
Matrikelnummer 0703588

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Wien, 11.04.2017

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Code Generation for fUML

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Ulrich Fischer
Registration Number 0703588

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Vienna, 11.04.2017

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ulrich Fischer
Nevillegasse 2/23, 1050 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 Verfasser)

Danksagung

Ich bedanke mich beim BIG der TU Wien für die Möglichkeit, meine Arbeit in dieser Forschungsgruppe abfassen zu können, bei meiner Betreuerin Tanja für ihre unermüdliche Unterstützung und besonders bei meiner Familie – Mama, Papa und Bruder – die mich bei meinem Weg durch dieses Studium begleitet hat.

Abstract

The concept of abstraction is the foundation of the evolution of computer science. Complex concepts and techniques are composed to a level of abstraction on which new techniques can be built. The evolution of programming languages is no exception. The assembly language was an abstraction of binary machine code instructions, making them interpretable by humans. Up to the nineties, many programming languages required the user to manually allocate and deallocate memory—a task that is carried out by all modern languages without the knowledge of the user. Model-driven engineering (MDE) is a software development methodology that introduces models as the central element, raising the level of abstraction beyond programming languages. Software systems described by the means of modeling languages have certain advantages: they are easier to understand, better to maintain as the described system changes over time, and they can be translated to be used on different target platforms. The latter is carried out by model transformations and code generators.
UML is a standardized modeling language that is widely used to express the structure and behavior of systems. The transformation of systems described by means of UML concepts into executable code has been a tedious task for many years. This is because of the lack of formal semantics, which specify how models that conform to the abstract syntax of the modeling language have to be interpreted, which required the developers of code generators to rely on individual interpretations of the meaning of the modeling concepts. This circumstance was addressed with the fUML standard published in 2011. It covers the formal specification of the semantics of a subset of UML consisting of class modeling, activity modeling and action language concepts. Along with the standard, a conforming virtual machine was introduced that allows the user to execute fUML compliant models. However, up until today, no code generator compliant to fUML has been develped impeding the automated generation of implementation artifacts from fUML models for different target platforms.

The first goal of this thesis is to elaborate and implement a code generation approach that generates executable code from fUML compliant models. The main requirement of the generator is to produce code that, when executed, behaves equivalent to the fUML model from which it was generated. The second goal of this thesis is to develop a component that verifies the correctness of the code generated from an input model by comparing its execution to the execution of the model carried out by the fUML virtual machine. Both goals aim to provide support for one of the key features of MDE for fUML: the automated transformation from a higher level of abstraction to a lower level. Thereby, an increase in productivity and efficiency for users of fUML shall be achieved.

Kurzfassung

Das Konzept der Abstraktion ist das Fundament der Weiterentwicklung in der Informatik. Komplexe Konzepte und Techniken werden auf einer Abstraktionsebene zusammengefasst, auf der aufbauend weitere Techniken entstehen können. Die Geschichte der Programmiersprachen ist dabei keine Ausnahme. Die Assemblersprache ist eine Abstraktion binärer Maschinenbefehle, wodurch diese für den Menschen einfacher interpretierbar werden. Bis in die Neunziger mussten Benutzern Speicher manuell bereitstellen und wieder freigeben—eine Aufgabe die moderne Sprachen automatisch bewerkstelligen. Modellgetriebene Softwareentwicklung, englisch Model-Driven Engineering (MDE), ist eine Methodik der Softwareentwicklung, in der Modelle das zentrale Element darstellen und das Abstraktionslevel über jenes von Programmiersprachen heben. Softwaresysteme, die mittels Modellen beschrieben werden, haben gewisse Vorteile: sie sind einfacher zu verstehen, sind wenn sich das zugrunde liegende System ändert leichter zu warten und können übersetzt werden, um anschließend auf verschieden Plattformen eingesetzt zu werden. Letztere Aufgabe wird mittels Modelltransformationen und Codegeneratoren durchgeführt.
UML ist eine weit verbreitete, standardisierte Modellierungssprache. Sie wird genutzt um die Struktur und das Verhalten von Systemen zu beschreiben. Die Transformation von Systemen, die mittels UML beschrieben sind, zu ausführbarem Code, war lange eine mühselige Aufgabe. Das Fehlen einer formalen Semantik für UML, die angibt wie UML Modell zu interpretieren sind, erforderte es von Entwickler von Codegeneratoren eine individuellen Interpretationen der Bedeutung von Modellierungskonzepten vorzunehmen. Dieser Umstand änderte sich mit der Veröffentlich des fUML-Standards im Jahr 2011. Dieser Standard definiert die formale Semantik für einer Teilmenge von UML, bestehend aus Konzepten von Klassen- und Aktivitätsdiagrammen, sowie der UML Action Language. Gemeinsam mit dem fUML Standard wurde eine virtuelle Maschine eingeführt, mit der Benutzer fUML-konforme Modelle ausführen können.
Das erste Ziel dieser Diplomarbeit ist die Erarbeitung und Implementierung eines Codegenerators, der aus fUML-konformen Modellen ausführbaren Code erzeugt. Die wichtigste Anforderung ist dabei, dass sich die Ausführung des erzeugten Codes äquivalent zum fUML Modell verhält, von dem er generiert wurde. Das zweite Ziel ist die Entwicklung einer Komponente, die die Korrektheit des generierten Codes verifiziert. Dies soll durch einen Vergleich der Ausführung des generierten Codes mit der Ausführung des ursprünglichen Modells durch die virtuelle Maschine erfolgen. Beide Ziele streben an, eine wichtige Funktionalität von MDE für fUML bereitzustellen: das automatische Transformieren von einer höheren Abstraktionsstufe zu einer niedrigeren. Dadurch soll eine höhere Produktivität und Effizienz beim Arbeiten mit fUML erreicht werden.

Contents

CHAPTER 1

Introduction

1.1 Motivation

One could argue that every concept in computing is an abstraction of a much more complex concept underneath. The key press on a computer keyboard is an abstraction of the transfer of ones and zeros on a bus, which is an abstraction of electrons travelling on an electric wire. Simple mathematical instructions like add or subtract are an abstractions of sets of logic gates, which again are an abstraction of electronic switches and so on. Models are the same: they are used to express information in a more abstract way in order to learn, understand or simulate the subject the model represents. But first and foremost, models are a simplified or partial abstractions of reality. By that, models implement at least two roles [4]:

- *Reduction* feature: models focus on a certain detail while hiding details, which are not relevant in a certain purpose, and
- *Mapping* feature: models are based on an original individual, which is abstracted and generalized to a model.

In the simplest case, models, in form of sketches, act as drafts in order to communicate ideas and alternatives. But in the last decades, models played an increasingly important and more sophisticated role in the domain of software engineering [4].

The benefit of systematically utilizing models in the field of software development has been known for a long time: For instance, in the late seventies, Computer-Aided Software Engineering (CASE) was used to support the complex development of the Ballistic Missile Defense systems by the US Air Force [1]. The requirements document for such systems often contained thousands of requirements. Checking every requirement against all others was concluded to be such an enormous task that automation was desperately needed. By the introduction of an automated requirements engineering system, models, in form of Abstract System Semantic Models

(ASSM), acted as guidelines in the implementation process and exceeded their purpose of drafting tools. The developed solutions in the field of CASE, however, did not find overwhelming acceptance [53]. The quality of general-purpose graphical modeling languages and their mapping to underlying operating systems was inadequate. The amount and complexity of generated code, which was needed to compensate for the lack of the underlying platform support, was beyond the capabilities of translation technologies available at the time. This resulted in CASE tools being hard to develop, debug and evolve and in CASE itself having a relatively little impact in commercial software. During the eighties and nineties, procedural programming language were replaced by more expressive, object-oriented language like C++, JAVA and C#. The introduction of these language constituted another elevation in the level of abstraction. Equipped with high numbers of reusable functionality, provided by user generated libraries and language frameworks, the necessity of rewriting common and domain-specific services decreased drastically [53]. However, the enormous number of different libraries and frameworks as well as their ongoing development made it impossible for software developers to become familiar with next-generation technology in a decent timespan.

Model-driven engineering (MDE) aims to find remedy for the described complexity problems by introducing models as the central element in the software development process [53]. MDE describes the principle of elevating the level of abstraction beyond programming languages. In place of low-level programming languages, modeling languages are used to describe the requirements, the structure and the behavior of a systems to be built in form of models, which act as basis for the transformation into various target software artifacts (source code, database schemata, etc.) on a variety of platforms [13, 53]. This task is carried out by code generators, which play a crucial role in MDE: they drastically increase the productivity while shielding the user of potential peculiarities of a target platform.

1.2 Problem Statement

The Unified Modeling Language (UML) [44] is a general-purpose modeling language and became the industry-standard for software modeling. Despite its broad adoption and tool support, there exist today no standardized or broadly adopted code generators for UML due to ambiguities in UML's semantics. In order to translate a model into a target language in a consistent manner, a code generator not only needs to know the concepts of the modeling language, but also how to correctly interpret them, i.e. the meaning of a given element must be known unambiguously. A modeling language's concepts and the relationships between them are defined by the language's syntax. The meaning of the concepts are defined by the semantics. Having that in mind, the development of code generators for UML used to be a challenging task: the semantics of many modeling concepts were defined vaguely and ambiguously or just did not exist in version 1.5 of UML. In response to this criticism, the Object Management Group (OMG), which adopted and maintains UML, cleaned up with most of the ambiguities in the UML 2.0 standard [39]. The remaining ambiguities were encapsulated into semantic variation points, which preserve a certain degree of flexibility in using UML in different applications scenarios. For instance, the semantics of a certain concept may be defined loosely when used for documentation purposes,

or very strict when used for code generation. Thus, previous code generation approaches [6, 14] had to take this factor into account.
However, the semantics were still not defined precisely enough. Many specifications were interpreted ambigiously which resulted in UML tools being incompatible. The desire for a standardization of a precise semantics prompted the OMG to propose and later released the *Semantics of a Foundational Subset for Executable UML Models* (fUML) standard [47], which defines the precise execution semantics for a subset of UML. The motivation for the fUML standard is to precisely specify the foundational core of UML and provide the behavioral semantics of this core in a consistent manner. By that, OMG aims to address the mentioned difficulties and bundle innovations in the field of executable models. The semantics of fUML are defined operationally in terms of a virtual machine, which enables the execution of models compliant to fUML. To model the structure of a system with fUML, concepts from UML class diagrams are used, and concepts of UML activity diagrams are used to model the behavior of systems. Equipped with execution semantics specified in the fUML standard, it is possible to implement code generators that do not need to fall back on an individual interpretation of the semantics of UML, but can rely instead on the standardized behavioral semantics. However, to the best of our knowledge, no code generator for fUML exists to this day. The challenge in the implementation of a code generator for fUML is that the generator needs to guarantee to provide code that behaves in the same way as the execution of the model by the fUML virtual machine.

1.3 Aim of the Work

The goal of this thesis is to elaborate and implement a code generation approach that transforms fUML compliant models into executable Java code reflecting the semantics specified in the fUML standard. The code generator shall be complete, testable, and flexible wrt. the target language as discussed in the following.

Completeness. The focus of this work lies on the translation of behavioral UML concepts, because for the structural concepts mature code generators for various target languages already exist. Thus, the presented approach aims to support only a minimum set of UML class diagram concepts but to provide a complete support for all UML activity diagram concepts.

Testability. The code generator should generate code for fUML models that behaves equivalent to the fUML models, i.e., the execution of the generated Java code should behave equivalent to the execution of the fUML model carried out by the fUML virtual machine. The result of the generation process must be tested for its validity in a simple (preferably automatic) manner. Automated test suites should be employed to continually verify the validity of the implemented mappings between fUML and Java.

Flexibility wrt. target language. The code generator implemented in the course of this thesis produces Java source code. However, interoperability between different kinds of target systems is a key benefit of MDE [2]. Therefore, the code generator must be designed in a way that the target language can be quickly replaced by another language. This requirement is addressed by

using state-of-the-art template languages for the development of the code generator as well as a loosely coupled architecture.

1.4 Methodology

The focus of this thesis is on the development of two software artifacts, in particular, a code generator and a code verification framework. Hence, the design science methodology (DS) [60] was chosen as methodological paradigm for carrying out this thesis. DS is constructed for the discipline of information systems research. In the scope of this methodology the term *design* describes the research process as a set of expert activities that produces design artifacts. In accordance with the goals of this work outlined in Section 1.3, the following artifacts have been developed in the course of this thesis:

1. Code generator
    a) Conceptual mapping between fUML and Java
    b) Generic code generation templates for processing fUML models independent of the target language
    c) Java-specific code generation templates for generating Java code from fUML models
2. Code verification framework
    a) Monitoring component for collecting runtime information about the execution of generated Java code
    b) Trace comparison component for comparing the runtime information collected for the generated Java code and for the original fUML model code

The artifacts have been designed following the activities of the DS methodology defined by Pfeffers *et.al.* [48]. These activities have been implemented in this thesis as discussed in the following.

Problem identification and motivation. UML is currently by far the most widely used modeling language. A lot of research in the field of CASE and MDE lead to a new way of software development which propagates the idea that modeling languages excel classic programming language in developing, and maintaining software systems. The translation of systems described by a model into a classic third-generation language like Java is performed automatically by a code generator. To achieve this for UML, the execution semantics for every supported UML concept needs to be specified precisely in order to generate code in a consistent manner. This precondition was not met until the release of the fUML standard in 2011. Despite the emergence of fUML, code generators for UML compliant to fUML do not exist yet. Thus, the aim of this thesis is to elaborate and implement a code generation approach that transforms fUML compliant models into Java code reflecting the semantics specified in the fUML standard.

Analysis. After the identification of the problem addressed by this work, the first step consisted of an extensive elaboration of the current state of the art of code generation methods for fUML as well as Executable UML, and other behavioral modeling techniques. This also resulted in a better understanding of the fUML standard as well as the functionality of the fUML virtual machine.
The code generator is based on functionality provided by the Moliz[1] project, which provides an extension of the reference implementation of the fUML virtual machine based on the work of the Business Informatics Group of TU Wien. Therefore, a detailed study of the fUML virtual machine, and the Moliz extension was necessary to to start with the design and development phase of the thesis.

Design and development. The developed software solution can be roughly separated into two autonomous components: the first component is the code generator, which is based on the work of Benjamin Bosters. It is concerned with the processing of fUML models and mapping all model elements into Java expressions. Additionally to the mapping task, the generator processes the control flows and object flows of input models in order to generate code in the correct sequence, selects the correct data- and list-types corresponding to the source model elements and provides libraries for primitive data type functions. The second component verifies the generated code. It monitors the execution of the generated code and stores information about the execution. The correctness of the generated code is evaluated by comparing the runtime information captured about the execution of the generated code with the runtime information captured by the fUML virtual machine about the execution of the corresponding fUML model. The two components are used by a test suite, which tests the correct processing of all supported fUML concepts.
The starting point for elaborating the mapping from fUML to Java was the list of 27 predefined actions for which the fUML standard specifies precise execution semantics. The mappings are based on the Section *Annex A: Java to UML ActivityMapping* from the fUML specification [47], as well as the Section *11: Actions* from the UML Superstructure specification [42]. Another important reference for the mapping was the source code of the implementations of the fUML virtual machine, which helped to overcome ambiguities in the implementation details of the generator. For each mapping, input models were designed that cover possible generation results before implementing the respective mapping. The exemplary input models served as test cases and enabled a test-driven development of the mappings and the continuous integration of additional mappings.

Evaluation. The developed code generator has been evaluated by automatically comparing the execution of generated code with the execution of the input fUML model carried out by the fUML virtual machine. During the development of the fUML to Java mappings, a test suite was elaborated, defining a library of test cases that execute different scenarios of sample input models. This allows the evaluation of each supported fUML action under different conditions. To test the correctness and reliability of the code generation on a broader scope, a comprehensive case study was conducted in addition.

[1] Moliz, `http://www.modelexecution.org/`

The evaluation process is similar for every tested fUML model: the runtime information about the execution of an fUML models is collected by leveraging the Moliz extensions of the fUML virtual machine. The runtime information about the execution of the generated code is collected by means of aspect-orientated programming techniques. After both execution traces have been collected, they are both transformed into a common format and then compared using the model comparison framework EMF Compare [5]. All deviations between the two executions are documented in a standardized manner.

1.5 Structure of the Work

The remainder of this thesis is structured as followed:

- **Chapter 2: State of the Art**
  This chapter is concerned with state of the art methods of code generation. Generators for different kinds of UML behavior diagrams like activity diagrams and state machines are discussed in this section.

- **Chapter 3: Background**
  This chapter gives an overview of the Unified Modeling Language, the history of model driven development and executable models, and an introduction to the fUML standard.

- **Chapter 4: Code Generation**
  This chapter describes the developed code generation approach. After an introduction to the general requirements, the developed code generation procedure is describes thoroughly. Finally, the detailed mappings of fUML concepts to Java constructs are presented and demonstrated based on examples.

- **Chapter 5: Code Verification**
  This chapter is concerned with the verification of code generated with the developed fUML code generator. The first part of the chapter gives a brief overview of the conditions for a generation result to be valid. The system responsible for monitoring the execution of generated code is introduced in the next part. The last part of this chapter is concerned with a detailed discussion of the method of verifying the equivalence of the runtime behavior of the generated code and the fUML model.

- **Chapter 6: Evaluation**
  This chapter describes the process with which the correctness of the generated code was evaluated. Furthermore, the results of the conducted case study and the developed test suite are presented.

- **Chapter 7: Conclusion and Future Work**
  In this chapter, the contributions of this thesis are summarized. After that, possible improvements and future work are discussed.

CHAPTER 2

State of the Art

The fUML standard was published in 2011 and lots of research activities are ongoing around fUML. Nonetheless, no implementation of a code generator for fUML models is available at the current date. However, lots of efforts were made in the domain of code generation based on other behavioral models, even UML activity diagrams conforming to a subset of UML overlapping with fUML. In the following, we give an overview of code generation approaches for UML activity diagrams and other UML and non-UML behavior diagrams.

2.1 Code Generation for UML Activity Diagrams

Gessenharter and Rauscher [12] describe an approach for code generation for UML activities preceded by model transformations. The presented code generator is able to generate code from the structural parts of a model, i.e., class diagrams, as well as code from sequences of actions. The implemented token flow concept supports the generation of code for actions with multiple return values resulting in a sequence of statements. Control nodes and guards, two concepts to control the token flow in sequences of actions, are also supported. The use of control nodes in the token flow increases the complexity of an activity. Especially the existence of more than one control node and cycles between two actions increase the complexity for the code generation dramatically and were not implemented in the prototype. Therefore, the generator is designed for activities with at most one control node between every of its actions. However, the paper also introduces a transformation method to simplify a model's activities to simple subsequences in order to successfully transform them to code. Every sequence of actions is translated into a sequence of Java methods. As sequences can be executed concurrently (e.g. as a result of a fork node), they are implemented as dedicated threads. The concept of *InterruptibleActivityRegions* (i.e. sequences, which leave the region on an edge that is interrupting the region) is realized by thread groups that can be aborted.
The model of the transportation system of an imaginary city was used to verify the presented code generation approach. The runtime characteristics of the generated code were compared to

those of two other activity interpreters that were results of previous works of the authors as well as to the runtime behavior of the fUML virtual machine. While this approach for generating code from UML diagrams is well elaborated, it does not fully comply with fUML. In particular, the token flow formalization by Sarsted [52], on which the presented approach is based, is not compliant with fUML. However, conducted tests show that the behavior of the generated code is equal to that of the fUML virtual machine.

The Fujaba (From Java and Back) project [35, 36, 61], developed by Nickel et al. since 1997, aims to provide visual programming languages by integrating UML class diagrams and a set of UML behavior diagrams. It's a software engineering tool that uses a combination of class diagrams, collaboration diagrams, state machines, and activity diagrams to provide round-trip engineering support for UML and Java. That means that Fujaba supports Java code generation from the diagrams as well as recovering the diagrams from the code. The structural part of a system is specified by class models. Activity diagrams and state machines are used to specify the behavior of a system. Any activity diagram or state machine can reference a collaboration model, which is used to describes the behavior of the activities. The collaboration diagrams act as notation for the graph rewriting rules, which are used to perform the code generation. With reverse engineering, Fujaba is equipped with a comprehensive tool to evaluate the code generation process. By comparing the source model with the reverse engineered model of the generated code, the correctness of the generated code can be verified. Compared to the code generator developed in this thesis, the behavior of the activity nodes defined in an activity diagram is modeled by the means of collaboration diagrams. Furthermore, the presented code generator only supports a limited subset of UML defined by Köhler et al. [22], for which precise semantics were available at the time.

Usman et al. [57] present a code generation tool, namely *UJECTOR*, that generates executable Java code from UML class diagrams, sequence diagrams and activity diagrams. The presented code generation approach first transforms class diagrams into Java code skeletons. Sequence diagrams are used to represent the flows of messages, which are translated into class methods. The sequence diagrams reference activity diagrams, in which behavioral actions are utilized to model object manipulations and user interactions. The generator maps the UML actions to Java code and places the result within the class methods. A case study is conducted in which the generation result of the presented tool is evaluated against the results of similar tools. The pursued goal of the UJECTOR project is very similar to the one presented in this thesis. However, the code generator in this thesis solely utilizes UML activity diagrams to model the behavior of a system. Furthermore, the fUML to Java mappings elaborated in this thesis are based to the formal semantics definition of the fUML standard, which was released in 2011. UJECTOR was presented in 2008 and therefore relies on the informal descriptions of UML's semantics given in the UML superstructure specification for the implementation of the mappings from UML actions to Java code.

2.2 Code Generation for Behavioral UML Diagrams other than Activity Diagrams

Ciccozzi et al. [7, 8] present a code generation approach that utilizes UML component diagrams, UML state machines, Alf code and, additionally, an intermediate model representation to produce code for a certain target language. Component diagrams are used to specify the structural aspects, and state machines and the Action Language for Foundational UML (Alf) [43] to describe the behavior of a system. The execution semantics of Alf, which is a textual representation for UML modeling elements, is given by mapping the Alf syntax to the syntax of fUML. In the presented code generation approach, it is used to enrich the behavioral description in order to reach the expressive power to produce executable code. No other action code in terms of code directly written in the target language is used to describe the behavior of the system. Alf defines three syntactical conformance levels, whereby the presented approach conforms to the minimal level. This decision is justified by the fact that concepts provided by the minimal set reflect the concepts found in the target languages. The use of an intermediate model representation, which is defined in Ecore, promotes the generic goals of the presented approach. While the presented code generator transforms the source model into C++ code, the intermediate model approach allows the reusability of model-to-text transformations for other target language. The code generation process is separated into two major steps: the transformation of the source model, consisting of a UML component diagram, a UML state machine and Alf code into the intermediate model and the actual generation of target code based on the intermediate model. The transformation of the behavior, defined in terms of Alf action code, into the intermediate model is carried out with QVT Operational [23], an OMG standard language that provides an intuitive way to specify model transformations. The transformation of the intermediate model into the target language is performed with Xpand [10]. The correctness of the generation results were validated against two comprehensive, industrial case studies.
The main difference between this approach and the approach presented in this thesis is the fashion in which the behavior of a system is defined in the source model and how the model is transformed into executable code. In this thesis, it is assumed that the behavior of a system is defined by means of fUML conform UML activity diagrams, while the approach presented by Ciccozzi et al. utilizes UML state machines with embedded Alf code. To provide a high level of flexibility in regard to the generated target language, Ciccozzi et al. propose the use of an intermediate model representation, while the approach presented in this thesis aims to provide such flexibility by separating the code generator implementation and the actual target language code.

Language embedding [9] is a method that allows the implementation of a language in terms of an existing language. Dévai et al. [9] present an implementation prototype of a modeling language, namely txtUML, which stands for *textual, executable, translatable UML*, that uses Java as host language to develop, execute and debug a system described in txtUML and transform it into different target languages. The host language provides an API that offers the features and construct of txtUML. Model visualization is provided for Papyrus [24]. In its current version, txtUML supports classes to define the structure and state machines and Java code, that acts as

action code, to describe the behavior of a system. The model, since defined by the means of the provided Java API, can then be compiled, executed and debugged like a regular Java program. The execution semantics are based on xtUML [62], an extension to UML based on the Shlaer-Mellor Method of MDA [54, 55] and extended by UML 2 abstractions. Abstractions covered by other specification like fUML [47] and PSCS [46] are adopted. Unavailable executions semantics for e.g. state machines are substituted by informal semantics provided by the UML 2 standard. The approach provides a model compiler that facilitates the transformation of a model into a specific target language. A prerequisite for this is that the model is in a queryable form. Java reflections and AspectJ are used to build an Ecore-based representation of the model, which then serves as basis for the code generation (and other features like visualization). The current prototype supports the generation of C++ code.
The virtual machine, provided by the fUML standard [47], is related to the presented approach in the sense that is uses Java as a host language to execute models defined in an embedded language and, by that, provides the execution semantics in an operational fashion. However, since the proposed method utilizes UML state machines and a restricted subset of Java to describe the behavior of a model, no direct relation to fUML is given.

The HUGO project [21], introduced by Knapp and Merz in 2002, is a project that provides a code generator that produces Java code that behaves as prescribed by the state machine of an input UML model, as well as a model checking mechanism that verifies the consistency of UML state machines against specifications expressed as collaboration or sequence diagrams. The generator transforms every state of the state machine into a separate instance of generic class that provides the standard runtime component state for UML state machines. The objects of these classes provide methods for activation, deactivation, initialization, and event handling and are complemented by the generated code that is specific to the given model, in a next step. For every UML class in the input UML model that contains a method body, a separate Java class is generated. Furthermore, the generator produces an event queue and an event dispatcher for every state machine. The transition from one state to another is implemented by a greedy algorithm as proposed by the UML standard. Except time and change events, all features of UML state machines are supported by the presented code generator approach.
The focus of the model checking component of the HUGO project is to verify the consistency of UML state machines against given interaction diagrams (collaboration and sequence diagrams), that describe a single system run. In total, three different approaches are presented; the first and second approach are based on the model checking tool SPIN [26]: The first implementation of the model checking back end compiles collaborations into observer automata, which synchronize on messages transferred within an interaction. This reduces the feasibility of the interaction to a reachability problem for the observer automaton whereby a successful run results in a counter example. This approach, however, showed itself inefficient although it promised well in regard to the validity of the analysis. With a variety of optimization measures applied, more than half a million states and transitions where generated for a small sample model taking SPIN more than a minute to analyse. The second implementation addressed the performance issues by replacing the generic translation process from the UML state machine model into the input language of the SPIN model checker by a translation processes tailored to a given model.

These measures, based on the ideas presented by Lilius and Porres [27], reduced the duration of the analysis to under a second. The third approach utilizes UPPAAL [25], a tool to model, simulate and verify real-time systems, which made the analysis of models involving real-time constraints possible. In its strive to adhere to the UML semantics, the HUGO project is very similar to the approach presented in this thesis. In order to conform to the UML semantics, the HUGO project uses a standard runtime component state, consisting of classes that are organized along the UML metamodel, while the generator presented in this thesis addresses this requirement by employing the fUML semantics. Furthermore, HUGO does not provide support for UML activity diagrams.

The code generation approach by Chauvel and Jézéquel [6] presented in 2005, uses UML state machines to describe the behavior of systems and is also concerned with the fuzziness of the UML semantics at the time. The authors describe a code generation approach that utilises the concept of semantic variation points. Semantic variation points constitute an intentional degree of freedom for the interpretation of the modeling language's semantics. The paper focuses on three aspects of UML state machines, which make use of semantic variation points: time management (synchronous vs. asynchronous), event selection policy, and transition selection policy. It would be conceivable that, for example, in terms of time management, a library business application modelled with a state machine is expected to behave differently from a CD-player in a real-time system modelled in terms of a state machine. Semantic variations in these aspects are expressed in form of semantic metamodels, which act as objectification of the part of the semantics that is subject to variability. Object-orientated features like inheritance and delegation are used to model the variabilities. The paper also presents an approach to model non-functional implementation variations, such as memory footprint, flexibility and maintainability. Together with a source UML model, they form the platform for the generation, which, in the sense of MDA, can be seen as a platform independent model. By applying the defined semantic variations and the defined implementation variations on the source model, the code generator generates a target model where all semantic and implementation choices are made explicit. However, rather than introducing a code generator for a specific target language, the presented approach presents a way to uncouple implementation choices and semantics issues from the code generation task.

In 2005, Long et al. [29] presented a code generation approach, in which the code generator produces class skeletons and method signatures from class diagrams and the method bodies from sequence diagrams. In a UML based development process, different kinds of UML diagrams are used to represent the artefacts created in a certain phase of the development process. This provides different views on the modelled system, whereby every view focuses on different aspects of the system. These models, however, are confronted with the problem of consistency among the different views of the whole system. The consistency checking problem, on which the paper focuses, emerged from the fact that the syntax and semantics of UML was defined informally and imprecisely at the time. As described in Section 3.3, the fUML standard, which covers the formal specification of the semantics for a subset of UML, was not released until 2008. To provide consistent code generation, the code generated from a model of the system should include optional modeling features like role names in class diagrams or object names in

sequence diagrams. In the approach by Long et al., this requirement is addressed by the usage of the object orientated language Relational Calculus of Object Systems (rCOS). It is equipped with an observation-oriented semantics and a refinement calculus. After a formal definition of the programming language and the definition of consistency requirements, the paper investigates how to formalize class diagrams and sequence diagrams and how they are transferred into rCOS code. An algorithm to check the consistency of sequence diagrams and class diagrams is presented in detail. If both models are found to be well-formed, the presented code generation approach translates the input models into rCOS code by traversing through the sequence diagram. The semantics of the generated code, defined by Yang et al. [63], is proven to formally comply to the semantics of UML models by Lui et al. [28].

Engels et al. [11] proposed a code generator for extended UML collaboration diagrams that is not only able to generate class definitions and operation signatures but also to generate the procedural flow within the operations. The goal of the generator is to transform the functional behavior of a system into executable code fragments. In collaboration diagrams, messages between instances of classes are used to describe the structural context and the behavior of interactions. They can be deployed in a variety of different scenarios and project phases. In early phases, collaboration diagrams can be used to describe the functionality of an operation defined in use cases on a high level of abstraction. In a method-orientated usage, collaboration diagrams can be employed to describe the functionality of an operation of a system on an abstract level. In the scope of the presented generation approach, one collaboration diagram serves as source for exactly one target method.
The authors explicitly favour collaboration diagrams over sequence diagrams as source models for the transformation process, since they inherently hold important information for the generation of code, e.g. how objects can be accessed and how they are transported between methods. Because of the purely object-orientated nature and the support for concurrent programming, Java was selected as target language of the transformation. A set of pragmatic guidelines and constraints are introduced as extensions to UML 2 collaboration diagrams to ensure that the translation into the corresponding Java code can be performed automatically. Input models that follow the guidelines are considered well-formed. Before a well-formed source model is processed by the generator, it is transformed into a model conforming to a refined UML metamodel. To reflect the required assumptions and constraints for a well-defined model, certain metamodel elements have been added, while others have been modified or removed. The generator is based on a pattern-based transformation algorithm and uses sets of rule schemata and patterns to perform the actual generation of Java code. The approach is based on a compiler construction method known as two-level grammar [58], which allows the separation of a certain collaboration diagram and the generated code.
The paper shows that UML collaboration diagrams are potentially suitable to describe the behavior of a system. However, the lack of support for operations on primitive data types or predefined enumerations prevents them from being used as a visual programming language. A possibility to verify the correctness of the generated code in regard to the source model is not presented in scope of the paper.

Moreira et al. [34] and Vidal et al. [59] propose automatic source code generation in the domain of embedded systems. Their presented approaches translate UML models into the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [49]. Embedded systems are computing systems that are designed to perform only a few dedicated tasks. They usually consist of a hardware component upon which software application programs execute. The approach present in [34] allows the modeling and verification of VHDL code of an embedded system before synthesis tools translate them into real hardware. In the fist step, functional and non-functional requirements of the system are identified. In the second step UML class diagrams are used to specify the structure and sequence diagrams to define the method behavior of the functional requirements. To deal with the ambiguities of UML semantics, the specifications of the input models are translated into the Distributed Embedded Compact Specification (DERCS). In the last step, the generation process is completed by mapping DERCS elements to corresponding code fragments in VHDL. The approach presented in [59] uses UML class diagrams, state machines and a subset of C++ as action language to define an application model, which specifies the behavior and functional architecture of a system. Together with a platform model and an allocation model, the system is described on the Detailed Modeling Level (DML) on which the automatic generation of VHDL code is performed.

2.3 Code Generators Provided by Commercial UML Tools

A wide range of commercial products for the generation of Java code from structural and behavioral UML diagrams are available. A selection of these is described briefly below. Many of them support code engineering from a variety of input modeling languages into different target languages. However, the described features below are limited to code generation from UML models into Java code and reverse engineering UML models to Java code.

MagicDraw [37] by No Magic, Inc. is a visual modeling tool that supports round-trip engineering between UML class diagrams and Java code. Not every feature of the Java language can be directly mapped to UML elements. In order to cover missing Java features, MagicDraw uses UML stereotypes and self specified properties. The creation of *Code Engineering Sets* allows the user to associate a UML class diagram with a Java file whereby every modification in the diagram is immediately reflected in the Java file and vice versa. The translation of behavioral diagrams into executable Java code is not possible. However, a visualisation feature supports the creation of sequence diagrams based on Java methods.

Like MagicDraw, *Rational Rhapsody* [16] by IBM® provides roundtrip engineering in the sense that the source model is always in sync with the generated code. It does support code generation from behavioral models. Sequence diagrams are used to model the data exchange between objects while UML state machines are used to model the lifecyle of objects. However, Rhapsody does not provide an own action language to describe the actual behavior performed in a certain state. In order to do that, the target language (i.e. Java, C, C++) is used. This results in a tight coupling between the platform-independent and the platform-dependant model.

Rational Software Architect (RSA) [17,18] by IBM® is a suite of software design, modeling and development applications, which are built on the Eclipse Platform. RSA supports the code generation from UML package and class diagrams to Java code. Additionally, the UML action

language (UAL) can be utilized in *OpaqueBehaviors*, *OpaqueActions* and *OpaqueExpressions*. This allows the definition of executable models in a platform-independent fashion. *Marking models* then provide the components to transform system models into a specific target language. UAL is based on the fUML standard [43] but it does not support all concepts.

Enterprise Architect (EA) [56] is a comprehensive modeling tool that provides Java code generation from UML class diagrams. Besides that, the code generation from three UML behavioral diagrams, including activity diagrams, is possible. To do so, all behavioral constructs have to be contained within a class. Before code from activity diagrams is generated, the input model is validated by a graph optimizer that analyses the model and transforms it into constructs from which Java code is generated. Due to missing documentation, it could not be verified whether the code generator for UML activity diagrams is conform to fUML.

2.4 Code Generation for Process Models

In the field of process modeling, Roser et al. [50] presented a code generator in 2007 that transforms models into executable workflow code. The presented solution provides a modeling language, a modeling tool and a code generator that allows the definition of domain specific languages (DSL) for process-orientated applications. The goal of the approach is to enable users with little experience about code generation to generate workflow code from abstract models.
In order to optimize solutions in the Enterprise Resource Planning (ERP) domain, the authors identify the flow of services, rather than the services themselves, as targets of necessary improvement measures. Hence, they seek to provide an option that enables enterprises to design components in a loosely coupled fashion, like service-orientated architecture (SOA). The presented tool, namely AgilePro, is based on SOA and allows users to model their business processes, and preview and execute them on a process engine. The processes are modelled as domain-specific models (DSM), which conform to a DSL. The metamodel of AgilePro extends the UML 2 metamodel for activity diagrams with information about responsibilities and functions. Model templates, which conform to the extended metamodel, are used to provide reusable modeling concepts for domain-specific modelling for ERP, Customer Relationship Management (CRM) and other fields of applications. In order to generate code from DSM the models are transformed into the Business Process Execution Language (BPEL) [38], an executable business process language. As a block-structured language, BPEL, however, does not support all concepts used by the higher-level process graphs. Therefore, model analysing and restructuring are necessary to map the graph to BPEL code. The presented model and code generation framework solve these challenges with a domain specific *Adapter for DSL Process Models*, which creates a representation of the input model in the common process model format, and the domain independent *Process Transformer and Optimizer* and *Process Visitor*, which restructure the process model into a BPEL conform block structure. In the last step, the domain specific *Code and Model Generation Template* is executed. The paper presents two case studies that illustrate the code generation process in detail but does not elaborate on how the validity of the generation result in regard to the semantics of BPEL is ensured.

2.5 Summary

As can be seen from Table 2.1, all 16 approaches generate code from UML diagrams, five of which support the code generation from activity diagrams. Nine of the presented approaches generate Java code, the commercial tools [16, 17, 37, 56] provide the widest range of target languages while four approaches [29, 34, 50, 59] are concerned with the generation of code in domain specific languages. However, none of these approaches is compliant to fUML. Only one exising code generation approach is related to the fUML standard, namely the one by Ciccozzi et al. [7, 8]. However, Cicozzi et al. did not build a code generator for fUML as targeted in this thesis, but for the textual surface notation of fUML defined in the Alf standard [43]. Furthermore, they generate C++ code and do not define a mapping to Java. In contrast, one main goal of this thesis was to elaborate a code generation approach for fUML that is flexible with respect to the target language.

Approach	*Input lang. / diagrams*	*Act. lang.*	*Target lang.*	*fUML compl.*
Gessenh. et al. [12]	CD, AD	-	Java[1]	no
Fujaba [35, 36, 61]	CD, AD, SM, CLD	-	Java[1]	no
UJECTOR [57]	CD, AD, SQD	-	Java	no
Ciccozzi et al. [7, 8]	CD, CPD, SM	Alf	C++	yes
Dévai et al. [9]	CD, SM	Java	C++	no
HUGO [21]	CD, SM, SQD, CLD	-	Java	no
Chauvel et al. [6]	SM	-	-[1]	no
Long et al. [29]	CD, SQD	-	rCOS	no
Engels et al. [11]	CLD	-	Java	no
Moreira et al. [34]	CD, SQD	-	VHDL	no
Vidal et al. [59]	CD, SM	C++	VHDL	no
MagicDraw [37]	CD	-	Java i.a.[1]	no
Rat. Rhapsody [16]	SM, SQD	-	Java i.a.[1]	no
RSA [17]	CD, PD	Alf	Java i.a.[1]	no
EA [56]	CD, AD, SM, SQD	-	Java i.a.[1]	no
Roser et al. [50]	CD, AD	-	BPEL	no

[1] Flexible wrt. target language

AD: Activity Diagram, CD: Class Diagram, CPD: Component Diagram, CLD: Collaboration Diagram, PD: Package Diagram, SM: State Machine, SQD: Sequence Diagram

Table 2.1: Overview of state-of-the-art code generation approaches for UML

CHAPTER 3

Background

This chapter gives an overview of the central technologies and design approaches this work is based on. The Unified Modeling Language and how it became the by far most widely use modeling language is briefly described in the first part. The efforts of gaining more value from high-level models in the software development process, which coined the term Model Driven Development (MDA), is described in the second part. Finally, an introduction to the executable subset of UML, referred to as Foundational UML or fUML, is given in the third part.

3.1 Unified Modeling Language

The Unified Modeling Language (UML) [44] is a standardized modeling language that is widely used to express the structure and behavior of systems. The Object Management Group (OMG), an international technology standards consortium, adopted the specification in 1997 and is responsible for its further development since then. The first version of the language emerged in 1995 from the object-oriented method *Booch method* by Grady Booch [3], the *Object-Modeling Technique* by James Rumbaugh and the design patter *Object-oriented Software Engineering* by Ivar Jacobson. The aim of the specification was the unification of the best practices in the fields of modeling language design, object oriented programming and architectural description languages. The standardisation facilitated the exchange of models between different tools and programs. It can be used to specify, design and maintain systems in every field of application domain.

James Rumbaugh et al. [51] identify several reasons why the Unified Modeling Language can be considered to be *unified*:

- Historical methods and notations: UML combines well recognized object-oriented methods and provides a clear definition, notation and terminology for each concept.

- Development lifecycle: UML supports modeling concepts for every stage of the development process beginning with the Requirements Diagram to the Deployment Diagram. which is the cornerstone for iterative, incremental deployment.

- Application domain: The UML was not devolved for a specific purpose, it is supposed to be used in every application domain. The systems might be large, complex, real-time, or distributed. Although UML does not claim to be the perfect language for every purpose, it is intended to be the best general-purpose language for a wide application area.

- The UML is intended to be used for systems implemented (or to be developed) in various kinds of programming languages and deployed on different target systems.

- Development processes: Just like a general-purpose programming language supports various kinds of programming styles, UML supports and underlies different development processes rather than describing a specific development process.

- Internal concepts: The introduction of UML's metamodel led to the discovery and representation of internal relationships among various concepts.

One of UML's most important objectives is the ability to exchange models between various modeling tools. This requires a formal definition of the semantics of modeling languages and notation. The UML standard meets this requirement by providing a formal definition of the abstract syntax of the modeling concepts, their attributes and relationships, an explanation of the semantics of every modeling concept as well as specifications of the notation elements.

Structure of the Specification

The UML specification itself is defined in a metamodeling approach. That means that a metamodel is used to describe how models conforming to UML have to be structured. Since the version 2.0, the UML specification is split in two complementary volumes[1]: The UML *Infrastructur* [41] and the UML *Superstructur* [42], which is based on the metamodel defined in the Infrastructur.

Infrastructur

The Infrastructur specification contains fundamental language concepts and is represented by the packages *InfrastructureLibrary* and *PrimitiveTypes*, that contains a set of predefined, primitive types that are commonly used. The *InfrastructurLibrary* contains the packages *Core* and *Profiles*. The *Core* package is a complete metamodel. It is either imported or specialized by other metamodels on the same metalevel. The package can be considered to be the kernel of Model Driven Architecture since it represents a common core for all descendants. It also provides a minimal, class-based modeling language, intended to be used by the Meta-Object Facility (MOF), to create metamodels for the description of more complex languages.

[1] In an attempt to simplify the specification, the UML Infrastructure and UML Superstructure have been merged into one document in the specification of UML version 2.5, which was released during the writing of this thesis.

The package *Profiles* defines mechanisms to customize metamodels for the use in a specific application domain or platform.

Superstructur

The Superstructure contains the user level constructs of the UML. The specification is divided into the parts *Structure*, *Behavior* and *Supplement*. The *Structure* section contains concepts to describe the structure of systems. The *Behavior* section contains concepts related to the behavior of systems and the interaction of its components. While the concepts of the *Behavior* section gives answers to the question, how something is interacting with each other over time, the concepts of the *Structural* section defines what is interacting. Therefor the concepts of the *Structural* section are the basis on which all behavioral concepts build upon.

Compliance Levels and Language Units

The broad scope of UML led to the introduction of the concepts of compliance levels [2]. Certain domains may require a specific set of UML features while the rest of the modeling concepts is not required. To address this requirement, UML was designed in a modular way. Language units wrap up related modeling concepts and are used to express and describe aspects of systems from a certain viewpoint. These language units are divided into layers called *compliance levels*. Every layers depends on the previous one and contains additional modeling concepts.

The UML defines four compliance levels. In the Infrastructure part of the specification, the number of compliance levels is reduced to two: The *Level 0* and the *Metamodel Contructs*. The *Level 0* comprises a class-based language that serves as basis for modeling common object-based systems. The Levels 1 - 3 are condensed to the *Metamodel Contructs* level, which adds a language to define metamodels. Being compliant to a certain compliance level implies the realization of all language units that are defined in that compliance level and the levels below since they build upon one another.

In the Superstructure [44] part of the specification, all language units are assigned to a certain compliance level. The tables 3.1, 3.2 and 3.3 give an overview about the content of the compliance levels.

Diagram Types

The diagram types supported by UML can be divided in structural and behavioral diagrams. Structural diagrams are used to describe the static structure of objects in a system. They do not change over time and do not contain details of the system's behavior, but can, however, contain connections to behavioral modeling concepts.
The behavioral diagrams are used to model the dynamic behavior of systems and how and based on which events the systems changes. In its current version [44] UML supports seven structural and seven behavioral diagram types which are briefly described in the following section.

[2]The concept of compliance levels has been removed with version 2.5 of the UML standard since they were not found to be useful in practice.

Language Unit	Matamodel Packages
Actions	Actions::BasicActios
Activities	Activities::FundamentalActivities
	Activities::BasicActivities
Classes	Classes::Kernel
	Classes::Dependencies
	Classes::Interfaces
General Behavior	CommonBehaviors::BasicBehaviors
	CommonBehaviors::Communications
Structures	CompositeStructure::InternalStructures
Interactions	Interactions::BasicInteractions
UseCases	UseCases

Table 3.1: Content of UML Compliance Level 1

Language Unit	Matamodel Packages
Action	Activities::StructuredActions
	Activities::IntermediateActions
Components	Components::BasicComponents
Deployments	Deployments::Artifacts
	Deployments::Nodes
General Behavior	CommonBehaviors::SimpleTime
Interactions	Interactions::Fragments
Profiles	AuxilliaryConstructs::Profiles
Structures	CompositeStructures::InvocationActions
	CompositeStructures::Ports
	CompositeStructures::StructuredClasses
State Machines	StateMachines::BehaviorStateMachines

Table 3.2: Content of UML Compliance Level 2

Language Unit	Matamodel Packages
Action	Actions::CompleteActions
Activities	Activities::CompleteActivities Activities::CompleteStructuredActivities Activities::ExtraStructuredActivities
Classes	Classes::AssociationClasses Classes::PowerTypes
Components	Components::PackagingComponents
Deployments	Deployments::ComponentDeployments
Information Flows	AuxilliaryConstructs::InformationFlows
Models	AuxilliaryConstructs::Models
State Machines	StateMachines::ProtocolStateMachines
Structures	CompositeStructures::Collaborations CompositeStructures::StructuredActivities
Templates	AuxilliaryConstructs::Templates

Table 3.3: Content of UML Compliance Level 3

Structural Diagrams

The different diagram types depicted in 3.1 are described below briefly:

- **Class Diagram**:
  Class diagrams are static structure diagrams that are used to describe the abstract model scheme of a system by showing its classes and their relationships. Classes are modeled as rectangles which are divided into three sections. The first section is reserved for the name and general properties (e.g. polymorphism and stereotype keywords), the second and third section are reserved for attribute and operation definitions, respectively. Attributes consist of a name, an optional visibility keyword ('+' for *public*, '-' for *private* and '#' for *protected*) and an optional type (e.g. 'String', 'int', ..). Additionally, multiplicity properties indicate how many occurrences of the attribute can exist in an instances of the class. Operations also consist of a name, an optional visibility keyword and an optional return type. A list of parameters is used to define inputs and results of the operation. It is important to mention that the behavior of operations can not be modeled within the scope of class diagrams. Classes can be connected with associations to model the relationships between them. Usually, associations connect two classes (binary associations), although the number of classes being part of an association is not limited. Directed association edges, cardinality indicators, association classes and other modeling concepts are used to increase the expressiveness of associations [15].

- **Object Diagram**
  Object diagrams are suitable to give an exemplary instantiation of a system. They illustrate the instantiation of class diagram at a certain moment. The diagram elements usually

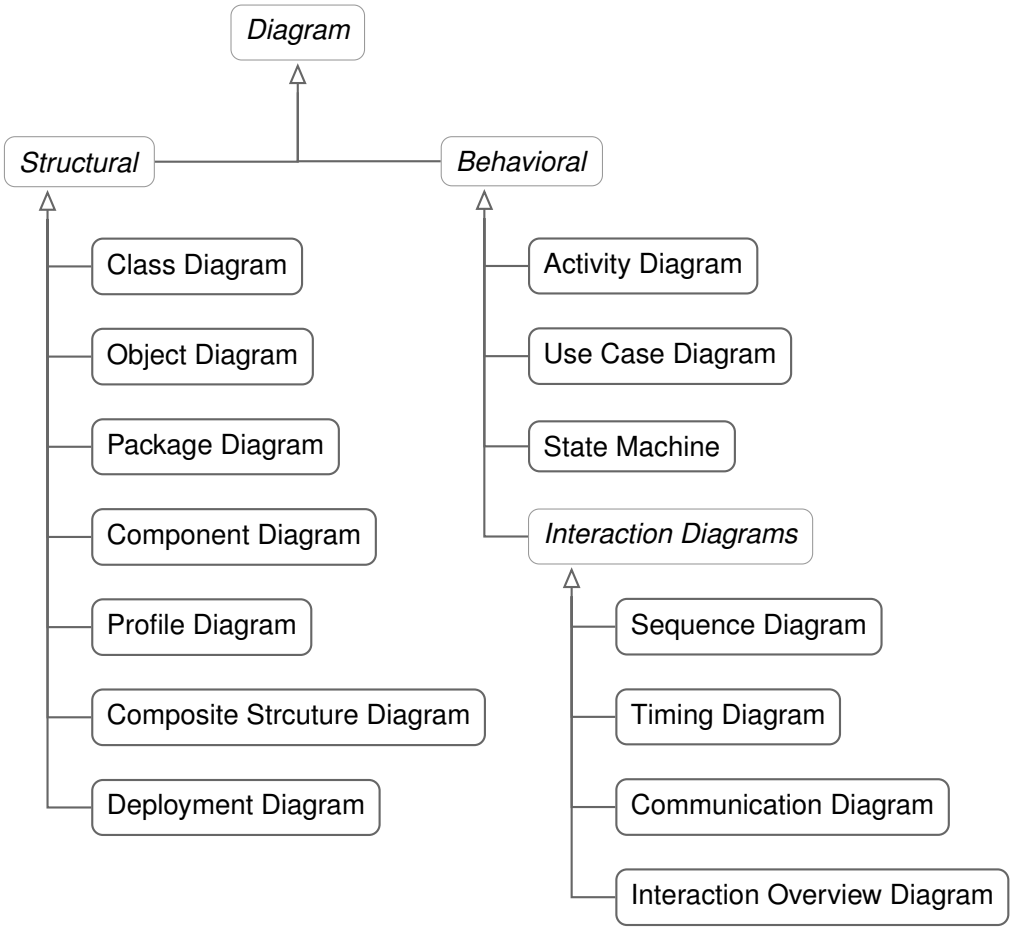

Figure 3.1: UML 2.4.1 diagram overview

contain information about the classifier, the instance name and the attribute values of the objects.

- **Package Diagrams**
  Packages are used to organize model elements into groups. They provide two important relation concepts which are used to implement the modular characteristic of the UML: *Package import* and *Package merge*. Package Diagrams usually come to use when systems exceed a certain size and need to be organized in several smaller ones.

- **Profile Diagram**
  Profile diagrams are used to create profiles on a metamodel level and provide a simple way to describe extensions of UML by using the *stereotype* class. The stereotype describes how an existing class is extended. This functionality can be used to create platform- and domain-specific terminology and notations.

- **Component Diagram**
  Component diagrams are mostly used to provide a physical view on a system, rather than a logical. They don't describe the functionality of a system but the components that are used to provide those functionalities. Component diagrams are usually deployed during the implementation and development phase of a project. A component itself is a structured class that represents a modular part of a system with encapsulated content.

- **Composite Structure Diagram**
  Composite structure diagrams are used to describe the internal structure of a class. *Parts* are used to illustrate the parts a class consists of. *Ports* and *Connectors* illustrate the connections of the parts and other elements within the class.

- **Deployment Diagram**
  Deployment diagrams model the physical environment of a system. *Artifacts* are used to represents a physical information element and *Nodes* depict physical devices and execution environments (like the operation system).

Behavioral Diagrams

- **Activity Diagram**
  Activity diagrams are used to model workflows and detailed behavior using activities and actions. Actions are elements which represent an atomic step within an activity. The sequence of process steps is determined by connecting actions via directed edges. Every action can have a set of incoming and outgoing edges that specify the control flow or, if objects are passed, the data flow. After an action is completed, its successor is being processed. If an action has more than one incoming edge, all edges need to be satisfied before the action is executed. Decision nodes are used to guide the flow in one direction or the other. To model a conditional behavior, outgoing edges of a decision node are equipped with guards. Merge nodes are used to combine multiple incoming edges. To model concurrent behavior, join and fork nodes are used to split and join the control and object flows.

- **Use Case Diagram**
  Use case diagrams are very suitable to clarify and define the requirements for software in early development stages and get a high level view of a system. The main modeling elements are *Actors* and *Use Cases*. The relationship between these elements are used to illustrates which groups of users make use of which sets of functions of a system. *Include* and *Extend* associations depict the kind of relations between the use cases within a system.

- **State Machine**
  State machines are used to illustrate the behavior of systems in event-based way. The system is modeled as graph, whereas the nodes of the graph are represented by *States* and the edges by *Transitions*. States are defined as a condition in which an object exists. They can be source and target of any number of transitions. *Simple States* are the central modeling concept. By definition, they have no further sub states. Every state can have

a set of activities that are performed at its entry or exit, or while the system is in this state. Transition between states are used to describe state transitions of a system. They model how a system reacts (i.e. changes in it's state) to the occurrence of certain events. *Pseudostates* are used to realize complex state transition paths. Examples are initial and terminal states but also join and conditional fork states, which are used to split and merge transitions.

- **Sequence Diagram**
  Sequence diagrams are used to model the interactions between systems or parts of a system. The focus of the diagram lies on the sequence of messages between processes to show how they operate with each other and in which order.

- **Timing Diagram**
  Timing diagrams are used to model the behavior of individual classifiers and interactions of classifiers throughout a given period of time. The focus of the models lies on the timing of events that cause changes in the conditions of lifelines. Lifelines represent individual participants and changes in a participant's condition or state is represented by the steps in its lifeline.

- **Communication Diagram**
  Communication diagrams are used to model interactions between objects or parts. Lifelines represent individual participant in the interaction. Lines are used to model messages between lifelines while an arrow above a message indicates the direction of the communication.

- **Interaction Overview Diagram**
  Interaction overview diagrams are used to model the control flow of the interactions on a high level of abstraction. They are a form of activity diagram. The nodes of the diagram represent interaction diagrams, which can include sequence, communication, timing diagrams or again interaction overview diagrams.

3.2 Model Driven Architecture

The Object Management Group (OMG) adopted UML and the Meta-Object Facility (MOF) in 1997. The uncontrolled growth of specialized middleware solutions lead to the development of a MOF-based framework that is supposed to remain fixed while the infrastructure landscape around it changes over time. The result was MDA® [45], a framework that prescribes the development of applications based on a platform-independent model (PIM) of the application. This guideline leads to applications that usually consists of a PIM, plus one or more platform-specific models (PSMs) and complete implementations, one on each platform that the application developer decides to support. PIMs are used to describe the structure and the behavior of a system in a formal and platform-independent manner and are through the use of PSMs automatically transformed into software artefacts like scripts and source code on different platforms.
Model Driven Architecture follows the idea that modeling is a better foundation for developing and maintaining systems than writing code directly [32]. Following this central principle, MDA aims to provide support for every step in the life cycle of systems beginning from the specification of requirements to business modeling to actual implementations.

Basic Concepts

In this context, the term *system* can refer to all kinds of systems: a hardware compound, software, a company, a business process etc. The characteristic they all have in common is that they consist of components which are in certain relationship to accomplish some purpose.

A *model* is abstracted information that selectively represents some aspects of a system. The model is related to the system by an explicit or implicit mapping [45]. The information a model may represent can be of any kind of nature: software, hardware, and other domain-specific aspects of a system. These models can be expressed using any kind of UML diagram listed in Section 3.1 or any other modeling language.

The Value of Models

The essential goal of MDA is to derive benefits from models and modeling that help dealing with the complexity and interdependence of systems. Models provide a whole variety of benefits, the most important benefits in regard to this thesis are the following.

- **Eased communication**
  One of the most important benefits of models is their capability to help individuals to come to a common understanding of a subject or problem area. These models then can be used for the production of documentation, technology artifacts and executable systems [45].

  For instance, UML class diagrams are used to specify the structural features of a system and activity diagrams to define the behavior of the system. This is a big advantage for project stakeholders who often struggle to contribute with their important domain knowledge during the planing and development phase of an information system due to its very technical environment. By using these two basic UML diagram types, users without tech-

nical knowledge or even programming skills are able to participate in the early development stages of information systems.

- **Simulation and execution**
  MDA also promotes the idea to execute and simulate models. Models as data can drive simulation engines that can assist in both analysis and execution of the designs captured in models. Simulation assists in the human understanding of how a modeled system will function and is a way to validate that models are correct [45].

  Together with the fUML standard, a virtual machine was introduced, that lets users run and analyze fUML compliant UML models.

- **Automated transformation**
  Another important value of models is the possibility to fully or partially automate the derivation of artifacts and implementations from models. Such automated processes reduce time and cost to realize and maintain a systems and guarantee consistency across different target platforms [45].

  To our best knowledge, there is no code generation approach for fUML available to this date. This thesis aims to close this gap and provide an approach for automated transformation of fUML models to executable Java code.

Model transformation and execution are the two primary approaches described in MDA to automate the path from models to executable systems. Model transformations use models as input and transform them into artifacts, such as executable code, by using a transformation pattern. For example, a UML sequence diagram could be transformed into Python code, a BPMN model into an executable shell script. Model execution on the other hand uses model execution engines that accept models as input and directly executes them.

Both methods provide an important step. They enhance a model from a descriptive to a prescriptive information representation. What was specified in model form becomes usable as an executable system. However, for the transformation to be performed successfully, the models are required to be sufficiently detailed and accurate such that fully functional code can be generated for the software systems expressed in the models.

3.3 Foundational Unified Modeling Language

The continuous development in the field of MDA and executable models and the desire to use UML as a programming language led to the development of an OMG standard that defines execution semantics for a selected subset of the UML 2 metamodel and acts as a foundation for higher-level UML modeling concepts. This subset is referred to as Foundational UML or *fUML*.

Until the release of the version 1.5 of UML in 2005, the support for the expression of the behavior of a system by means of UML was very limited. However, with the adoption of action semantics in the version 1.5, an important tool to provide support for basic MDA principles like

model-based simulation and verification of system specifications and code generation was introduced. Based on action semantics, Stephan Mellor proposed the development method xUML, which describes the behavior of systems with executable models detailed enough to transform them into source code [33]. In xUML, UML state machines and a separately developed action semantics were used to describe the behavior of a system.
The description of the semantics of the elements of UML was incomplete, inaccurate and dispersed throughout the specification which lead to a simplification of the specification in the version 2.5 [44]. However, the biggest difficulty for users and tool developers was the lack of formalized semantics; many concepts of the UML are defined informally in English, leading to ambigious interpretations which result in incompatibility of UML tools. As as result of this critisims and in the response to the desire for a standardization of a precise semantics for UML, the proposal for the *Semantics of a Foundational Subset for Executeable UML Models* was requested in 2005. In 2008, the first version of the standard was adopted by the OMG and published in 2011. The fUML standard [47] covers the formal specification of the semantics of a subset of UML consisting of class modeling, activity modeling and action language concepts.

Boundaries of the Subset

The fundamental purpose of fUML is to serve as an adapter between the UML elements used for modeling and a target language on a specific platform. The definition of the fUML subset was performed with regard to three basic premises:

- **Compactness.** The extent of the subset should be as small as possible to easily facilitate the definition of a complete semantics and development of execution tools.
- **Ease of translation.** The transition from surface UML elements to fUML and from fUML to target platform languages should be as straightforward as possible.
- **Action functionality.** The specification only specifies the execution semantics for UML actions as they are currently defined with primitive functionality.

In many cases the premises *Compactness* and *Ease of translation* contradict each other and are the sources of tension. Features that are available in surface UML might have a corresponding feature in a specific target language. To ease the translation, it would be tempting to directly translate the UML feature into the corresponding target platform feature. However, for the sake of compactness, the feature might be removed from the intermediary fUML layer. In this case the feature needs to be translated into a set of coordinated UML actions that produce the same result as the removed surface feature. Furthermore the fUML-to-platform translation need to recognize the set of actions to map the desired feature.

Under consideration of this conjuncture, the specification resolves the choice between compactness and ease of translation based on judgements about which functionality between UML actions and platform features are used more widely than others. The features are separated into three groups:

- Functionality that is widely used both in UML and target platforms are supposed to have a one-to-one translation to the intermediary fUML subset. An example for that would be

classes with properties and operations that are essential elements also in surface UML and object oriented target languages.

- Moderately used features are not supported in the fUML subset directly but still should have straightforward translation support. Examples for that group are composite structures and simple state machines.

- Less used functionality that is in common of UML and platforms, which may have a complex transformation, are not included in the fUML subset.

The fUML standard contains the foundational core consisting of UML class modeling concepts to define the structure, and UML activity concepts, to define the behavior of systems. Thus, the abstract syntax definition of fUML corresponds to a strict subset of the UML metamodel.

fUML Subset

The fUML specification covers a subset of the UML metamodel to define the structure and the behavior of a system. The subset, depicted in Figure 3.2, contains the structural kernel of UML, which contains the UML package *Classes*, the behavioral kernel of UML, consisting of the UML package *CommonBehaviors*, a subset of the UML package *Activities* and a subset of UML's action language (package *Actions*). Thus, to describe a system within fUML, class modeling concepts are used to describe the structural features, and activity modeling concepts to describe the behavioral features.

Structure

As depicted in Figure 3.2, the central modeling elements provided by UML and included in the fUML subset to describe the structure of a system is a class (metaclass *Class*). Classes can own attributes (metaclass *Property*), which are of a certain type and multiplicity, and Operations (metaclass *Operation*), which can specify a set of parameters (metaclass *Parameter*). Relationships between classes are described with associations (metaclass *Association*). With Integer, String, Boolean and UnlimitedNatural, a set of primitive types (metaclass *PrimitiveType*) is provided.

Behavior

Concepts of UML activity modeling available in fUML due to the inclusion of the package *CommonBehaviors*, are used to model the behavior of a system. An excerpt of the metamodel of this package is depicted in Figure 3.3. Activities (metaclass *Activity*) constitute the central modeling element. They contain sets of activity nodes (metaclass *ActivityNode*) and sets of activity edges (metaclass *ActivityEdge*). Activity nodes can be divided into three types: Object nodes (metaclass *ObjectNode*), control nodes (metaclass *ControlNode*) and actions (metaclass *Action*). Object nodes are used to specify the input and output of activities and actions. They define parameter nodes (metaclass *ActivityParameterNode*), and input pins (metaclass *InputPin*) and

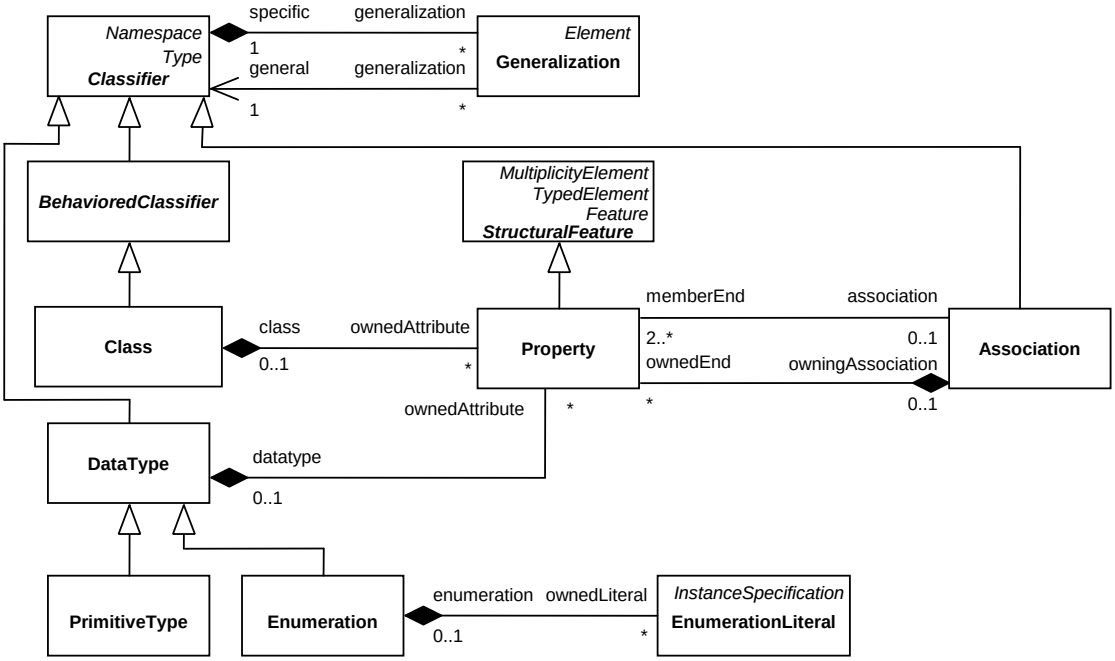

Figure 3.2: Excerpt of the fUML metamodel used to describe the structure of a system [30]

output pins (metaclass *OutputPin*). Control nodes are used to split (*ForkNode*, *DecisionNode*) and join (*JoinNode*, *MergeNode*) the flow of executions and define the start and end of activities.

Actions are the fundamental and smallest possible unit of executable behavior in fUML. The UML action language specifies a predefined set of 27 actions, that is contained in fUML. They can be divided into five different types. Object actions are used to create and destroy objects and perform specific tasks on instances of objects. Structural feature actions are used to add and remove properties of objects as well as perform read and write operations on object properties. Link actions are used to handle relations between objects. Communication actions are used for invoking activities. Structured activity nodes provide basic control statements like loops (metaclass *LoopNode*) and conditional expressions (metaclass *ConditionalNode*).

Semantics

The execution semantics of fUML are defined within the fUML execution model in an operational manner. The model itself is written in *base UML* (bUML), a subset of fUML, and defines a *virtual machine* which is able to interpret and execute fUML models. The behavior of the virtual machine is represented in Java whereby an translation into bUML is provided. The *Process Specification Language* (PSL) [19] is used to describe the semantics of bUML in a translational way. A reference implementation of the virtual machine, implemented in Java by Model Driven Solutions on account of the Lockheed Martin Corporation, is available under the Academic Free License. In general, the intention of the implementation of the virtual machine is to provide a tool to evaluate the conformance to the fUML standard, and by that, encourage potential con-

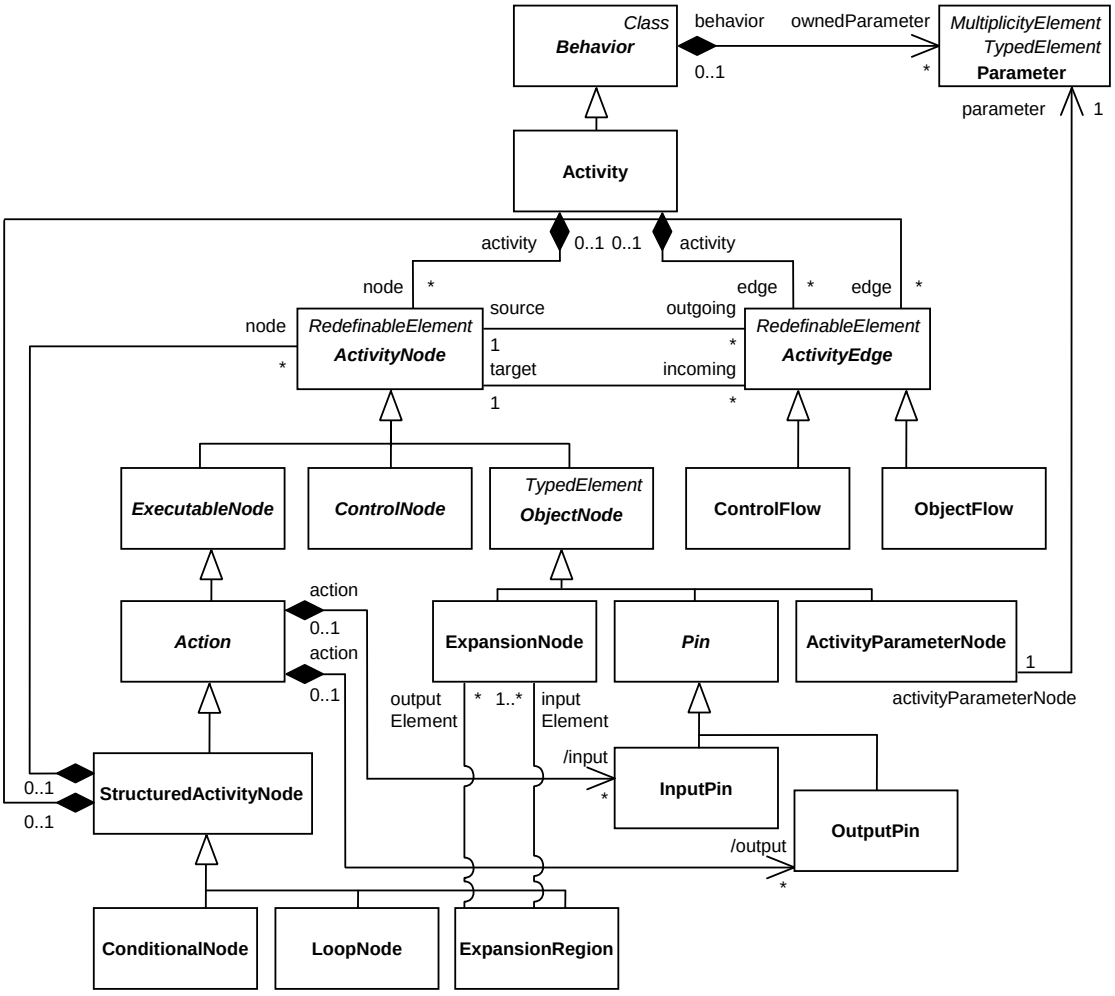

Figure 3.3: Excerpt of the fUML metamodel used to describe the behavior of a system [30]

tributors to use the fUML standard. In the context of this work, this reference implementation is used to verify the correctness of generated code in regard to a given input model as discussed in Chapter 4.

The execution semantics of fUML concepts are defined using a visitor pattern. Every metaclass of the fUML metamodel has a corresponding *semantic visitor class* that specifies its behavior. The execution model uses three types of semantic visitor classes. *Evaluation visitor* classes are used to define how values are created from a value specification. The evaluation visitor class for e.g. the type Integer defines how the specification of an Integer value (metaclass *LiteralInteger*) is evaluated to an Integer value (semantic visitor class IntegerValue). To define the semantics of activity nodes, *Activation visitor* classes are used. For instance, the activation visitor class AddStructuralFeatureValueActionActivation, specifies the semantics of the add structural fea-

ture value action (metaclass *AddStructuralFeatureValueAction*). *Execution visitor classes* are used to specify how instances of the meta class Behavior, i.e. activities, are executed. The definition is given operationally by the implementation of the execute() method of the visitor class.

The execution environment, which is also provided by the fUML virtual machine, is concerned with the execution of an fUML model. The Executor class is used to execute fUML models and servers as an interface for the fUML virtual machine. It provides operations to synchronously (operation execute()) and asynchronously (operation start()) start the execution of a behavior, as well as an operation to evaluate a value specification and return it's resulting value (operation evaluate()).

CHAPTER 4

Code Generation

This chapter first describes the requirements imposed on the fUML code generator developed in this thesis, then elaborates the technologies used for the implementation of the code generator, and finally describes the functionality of the code generator with code samples and example models. The description of the developed code generator is devided into three parts: code generation for class diagrams, code generation for activity diagrams, and code generation for actions. The developed software artefacts are available online. Further information about the public repository are provided in Appendix A.

4.1 Requirements

A code generator's task, in general, is the translation of a system described in an abstract, intermediate representation, into a target language, that can be executed by a machine. The code generator's task, in the scope of this thesis, is to translate fUML compliant models into executable Java code. fUML models consist of class diagrams which are describing the structure of the system, and activity diagrams, which model the behavior of the operations defined in the class diagrams. First and foremost, the code generator needs to generate code that behaves equivalent to the execution carried out by the fUML virtual machine. Besides that, the code generator shall be complete, testable and flexible wrt. the target language.
The requirement *Completeness* is addressed by aiming to provide an fUML to Java mapping for every of the 27 predefined fUML actions. Not all features provided by fUML are available in the specific target languages. For example, the fUML action `ReadExtent` is an action that, per definition, retrieves all current instances of a classifier. Java does not contain a functionality that provides this behavior innately. While it is not too complex to provide functionality for this specific missing action, the example reveals the challenges of implementing an fUML code generator since it is not possible to translate fUML actions to Java one-to-one. All supported actions and their mapping to Java code are presented in Section 4.5. In this section, also limitations and actions with limited support are described as well as unsupported actions.

The requirement *Testability* is addressed by two measures: Firstly, the fUML to Java mappings are developed in a test-driven approach. Before a mapping is implemented, a model is designed that contains the fUML action for which the mapping is implemented. Test models usually contain multiple test activities to cover different conditions under which the fUML action can be executed. Secondly, a test suite not only evaluates the correctness of the last implemented mapping but all previous mappings by re-generating the Java code for all test models and evaluating their execution against the execution carried out by the virtual machine.
Flexibility wrt. the target language is achieved by utilizing state of the art model-to-text technologies that provide a clean separation of the code of the code generator and the target language code. This allows the elaborated code generator implementations to be reused in order to elaborate mappings for a target language other than Java.

4.2 Model to Text Transformations

Regular programming language are not intended for the purpose of generating code. Dynamic and static code gets mixed up, functionality is hard to reuse and complex string concatenations often result in incorrect and improperly formatted outputs. Model to text transformations are used to transform MOF-based models into text. They utilize templates to express these transformations. Templates consist of text fragments and embedded meta-markers, which act as place-holders that are evaluated during runtime by querying the input model. By this, static code, in form of text fragments, and dynamic code, in form of meta-markers, are separated. MOF Model to Text Language (MOFM2T) [40] is an OMG specification that can be used to transform MOF-based models into text. Acceleo[1], JET[2] and Xpand[3] are a few of the more popular implementations of the MOFM2T standard.
The code generator for fUML developed in this thesis is written with Xtend[4]. Xtend is a statically typed, high-level programming language for the Java Virtual Machine. Besides a number of useful features like Lamda expressions, active annotations, dispatch methods and operator overloading, Xtend fully supports the template engine Xpand since 2011. Xpand is a language specialized on code generation. It enables to define code templates where expressions can be used to assign values to placeholders within the templates. Features like multi-line support and white-space handling contribute in facilitating the code generation process.

Xtend comes with a dynamic dispatch feature that allows the call of the same method for objects of a common super class. The decision of the invocation of the appropriate (i.e. the most specific) method is performed at runtime rather than compile time. This feature is heavily used by

[1]The documentation of Acceleo is available online at `https://wiki.eclipse.org/Acceleo`, accessed 10-02-2017
[2]The documentation of JET is available online at `http://www.eclipse.org/modeling/m2t/?project=jet`, accessed 10-12-2016
[3]The reference for the Xpand syntax is available online at `http://git.eclipse.org/c/m2t/org.eclipse.xpand.git/plain/doc/org.eclipse.xpand.doc/manual/xpand_reference.pdf`, accessed 10-12-2016
[4]The documentation of Xtend is available online at `https://eclipse.org/xtend/documentation/index.html`, accessed 10-02-2017

the code generator developed as part of this thesis. It comprises a set of generate dispatch methods, one for each supported fUML action. In each method, the template expression feature is used to provide readable string concatenation. Template expression are enclosed by triple single quotation and contain the Java code templates. The templates usually contain interpolated meta markers, enclosed with guillemots, which are replaced with variable values.
Listing 4.2 displays the simplified generate method for the action CreateObjectAction. The unique variable name for the newly generated object is added to the collection vars, subsequently the static code templates are completed with values from the expressions. Finally, the generation process continues by invoking the method handleOutgoing.

```
// Global map of variables and generated variable literals
var HashMap<String, String> vars;

// Global variable to provide unique variable names
var varCount = 0

// Generate method for the action CreateObjectAction
def dispatch String generate(CreateObjectAction a) {
   varCount = varCount + 1
   vars.put(a.name, "var"+varCount)

   '''
      // Generation of the Java code
      «a.classifier.name» var«varCount» = new «a.classifier.name»();

      // Continue code generation for action's outgoing edges
      «a.handleOutgoing»
   '''
}
```

Listing 4.1: Xtend-code of the (simplified) generation method for the CreateObjectAction

4.3 Code Generation for Class Diagrams

The code generation process can basically be separated into two phases. In the first step, the generator translates all elements that represent the structural features of the system, i.e. the class diagrams of an fUML model. In the second phase, all elements that describe the behavior of the system, i.e. the activity diagrams of an fUML model, are translated into executable code. In this chapter, we look at the first step of the process, which is the code generation for structural features defined in UML class diagrams.
The structural features of an fUML model are defined by elements of the types *Class*, *Property*, *Operation* and *Association*. The generator first processes all elements of the type *Class* and creates a Java-File with the same name as the class element. Polymorphism and nested classes are not supported, thereby only a simple class stub is being generated in the class file.
In the next step all property elements owned by the class element are processed. The type of a

	not unique	*unique*
not ordered	ArrayList	HashSet
ordered	ArrayList	ArrayList

Table 4.1: UML collection types

property is directly translated into the generated code. Ever property has a set of attributes, some of which are relevant for the code generation:

- The *upper* attribute is one of the two multiplicity elements that allows the user to define the cardinality of an element. If the upper property is set unequal to one (which is the default value), it indicates that more than one value can be assigned to the property.
- If the property is multivalued, the attribute *isOrdered* specifies whether the values are sequentially ordered or not. The default value is false.
- The attribute *isUnique* specifies whether the property allows duplicate values. If the property is not multivalued, this attribute is obsolete. The default value is false.

If the value of the *upper* attribute is greater than zero, the generated property is declared in form of a collection. The values of the *isOrdered* and *isUnique* attributes determine the type of collection used for the property. As displayed in Table 4.1, *ArrayLists*, which are ordered, provide random access and allow duplicates, are used in most of the cases. Since the generated code never relies on a collection to be unordered, it is used for both not ordered and ordered lists that allow duplicates. Since the Java Collection Framework does not contain an implementation of a unique collection that allows inserting and removing from a specified index, *ArrayLists* are also used for multivalued properties with *isOrdered* and *isUnique* set to true, whereby the uniqueness of the collection is ensured by checking if the list does contain the element to be added.
After all properties owned by the class elements are processed, the properties defined within *Associations* are processed. Every association can potentially connect an unlimited number of classes, which is expressed by one property for each connected class referred to as member ends. The code generator, however, only support binary associations, i.e. associations with two member ends. Every member end has the attribute *isNavigable* which indicates whether it can be accessed from instances at the other end. If a member end is set to navigable, the generator creates the code for the property on the other end of the association and thereby grants accessibility at compile time.
For every generated property, getter and setter methods are provided. The type, multiplicity, visibility and parameters of these methods are generated conform to the encapsulated property. In the generated code, properties are always accessed or modified via their getter and setter methods.
The last step of the translation of a class diagram is the translation of operations to method

stubs. Every operation can contain a set of parameters. For every parameter that has the *direction* attribute set to *in*, the code generator creates a parameter in the method signature. If the operation contains a parameter with the attribute *direction* set to *return*, the return type of the generated method is set accordingly. In fUML, additional parameter directions, namely *out* and *in out*, are available, which are not supported by the code generator. Lastly, the attribute *visibility* determines whether the access level of the generated method is private or public (the visibility types *package* and *protected* are not supported).

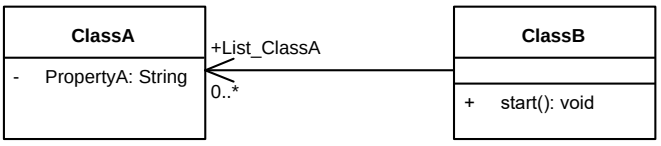

Figure 4.1: Sample fUML class diagram for illustrating the code generation

To illustrate the translation of the structural model, Listing 4.2 shows the generated code for the model shown in Figure 4.1. The diagram in Figure 4.1 shows a simple class diagram consisting of the classes ClassA and ClassB. ClassA owns the string property PropertyA. The two classes are connected with an association. The multiplicity attribute of the association end List_ClassA is set to 0..* indicating that one instance of ClassB is associated with zero or more instances of ClassB. The arrow models the navigable member end, in this case the property List_ClassA of the type ClassA. The multiplicity of property PropertyA is not set and therefore the default value 1 applies.

```
public class ClassA {

   private String PropertyA;

   public String getPropertyA() {
      return PropertyA;
   }

   public void setPropertyA (string PropertyA) {
      this.PropertyA = PropertyA;
   }
}

public class ClassB {

   public Collection<ClassA> List_ClassA = new ArrayList<ClassA>();

   public Collection<ClassA> getList_ClassA() {
      return List_ClassA;
   }

   public void setList_ClassA(Collection<ClassA> List_ClassA) {
      this.List_ClassA = List_ClassA;
   }
```

```
    public void start() {
    }
}
```

Listing 4.2: Code generation result for the activity shown in Figure 4.1

As a result of the code generation, the classes with correctly typed properties and operation stubs are created as can be seen in Listing 4.2. Getter and setter methods are generated for both private and public defined properties.

4.4 Code Generation for Activity Diagrams

In the second phase, the empty operation stubs are filled with code corresponding to the activities associated with the respective operation. Every operation is associated with one activity and every activity can only be associated with one operation.
In the first step, the generator creates a collection of nodes that are processed initially. The list contains elements of the type *InitialNode*, *ActivityParamterNode* and nodes with no incoming edges. For every item in the collection the *generate* method is called. Every node can have multiple outgoing edges that connect the node to its successor nodes. The generator processes the model by recursively calling the generate method of the successor nodes (and their successor nodes and so on). This process is repeated until there is no element in the list of initial nodes left. For every UML element supported by the generator, a generate method exists. A more detailed documentation of all supported elements can be found in Section 4.5.

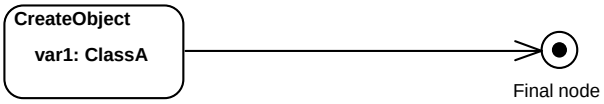

Figure 4.2: Sample fUML activity diagram for illustrating the code generation

Figure 4.2 shows an activity diagram consisting of three elements: A CreateObjectAction, a ControlFlow and an ActivityFinalNode. The generation process starts with the CreateObjectAction since it's the only one without incoming edges. After the code of the CreateObjectAction action was generated, the generate method of every outgoing element is called; in this case the generation method for the ControlFlow, which does not generate any code but calls the generate method of its target element. As a consequence, the last element, ActivityFinalNode is translated into code, namely into a return statement. Since there are no more outgoing edges left and the collection of elements to be generated initially is emptied, the generation process is finished. The result can be seen in Listing 4.3.

```
ClassA var1 = new ClassA();
return;
```

Listing 4.3: Code generation result for the activity shown in Figure 4.2

Listing 4.4 contains the simplified method of code generation of the activity diagram in pseudo code. The generations process starts by determine all initial nodes and assigning them

to a collection (line 3). Initial nodes are all nodes of the action that are of the type InitialNode, ActivityParameterNode or Action without incoming edges. The collection is then iterated (line 5), and then, depending on the type of the processes node, the correct generation method called (line 6).

```
1 function generateActivity(Activity a)
2 {
3    List nodesToProcess = a.getInitialNodes();
4
5    while nodesToProcess.hasNext() {
6       generate(nodesToProcess.getNext());
7    }
8 }
```

Listing 4.4: Function generateActivity of the code generator

A `generate` function, as shown in Listing 4.5, exists for every supported node of the generator. Xtend's dynamic dispatch feature invokes the appropriate generate function based on the type of the parameter n. If all incoming edges of the node have been registered by the Join Manager, the actual code generation (line 5), which is not shown in this sample, is conducted and the code generation continues by calling the function `handleOutgoing` (line 7). If the Join Manager is not ready, the current incoming activity edge is registered (line 10) and the process continues without any code being generated.

```
1  function generate(ActivityNode n)
2  {
3     if JoinManager(n).isReady() {
4
5        // <-- Code generation of node 'n'
6
7        handleOutgoing(n);
8     }
9     else
10       JoinManager(n).registeredIncoming++;
11 }
```

Listing 4.5: Generic generate function of the code generator

The function `handleOutgoing`, as shown in Listing 4.6, iterates over all outgoing edges (line 3) and all output pins (line 7) of the activity node n, and calls the corresponding `generate` function (lines 4, 8). By that, the code generation is continued along all outgoing edges of the node n.

```
1  function handleOutgoing(ActivityNode n)
2  {
3     while n.outgoings.hasNext() {
4        generate(n.outgoings.next());
5     }
6
7     while n.outputs.hasNext() {
8        generate(n.outputs.next());
9     }
10 }
```

Listing 4.6: Function handleOutgoing of the code generator

Control Flows and Object Flows

As described in Chapter 3.1, activity diagrams distinguish between two types of connections between nodes: *ObjectFlows* and *ControlFlows*. Control flows are used to direct the flow of control between activity nodes. The code generator simply iterates over all outgoing control flows of an activity node and calls the generate method of the control flow's target element.

Object flows, on the other hand, are connections between two activity nodes that transfer a value of a specified type. To transfer a value from one action to another, the value is placed on the output pin of the source action. The value is then transferred over the object flow and put on the input pin of the target action.

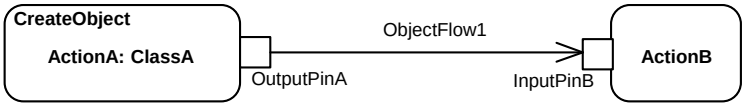

Figure 4.3: Sample fUML activity diagram illustrating the code generation for object flows

The code generator provides this functionality by keeping an internal list of variables (VARS) for every activity. Every time a value is being created or transferred between two nodes in the process of the code generation, a corresponding entry in VARS is added. The entry consists of the name of the node as key and the name of the generated variable as value. To avoid duplicate key entries, the node's name and its parent name are concatenated to form the key value if the node is an input or output pin. By looking up the value of the entry with the key of the source node's name, the access to the variable at the target of an object flow is given.

Figure 4.3 shows the transfer of an object from ActionA to ActionB. Since ActionB is not further specified, the generated code is equal to the code shown in Listing 4.3. Additionally to the generated code, an entry with the action's name `ActionA` as key and the generated variable name `var1` as value is added to the list of variables. The next node OutputPinA looks up the name of the variable in VARS using the name of its previously processed node `ActionA` and creates a new entry with its own name `ActionA_OutputPinA` and the looked up value `var1`. In the same manner, the next node InputPin adds an entry to VARS with the key `ActionB_InputBinP` and the value `var1`. The target action ActionB can eventually access the transferred

object by looking up the name of the generated variable using the the name of its input pin. If an action is processed multiple times due to, e.g. a loop node, potentially pre-existing values in the VARS list, which were added by previous processing of the action, are overwritten, which ensures that the subsequently processed node can only access the updated value. Listing 4.7 shows all key-value-pairs added to the VARS list for the example depicted in Figure 4.3.

```
ActionA: var1
ActionA_OutputPinA: var1
ObjectFlow1: var1
ActionC_InputPinC_1: var1
```

Listing 4.7: Key-value-pairs inserted in the variables list for the activity shown in Figure 4.3

The names of the generated variables are composed of the constant string `var` and an automatically incremented index.

Join Manager

Every action may have a set of incoming and outgoing control and object flows. The execution of the action won't start before all incoming edges are satisfied. According to the definition of token flow semantics of Petri nets, the action executes when there are sufficient tokens in all of it's incoming control and object flows. While every input pin can potentially require more than one token to be provided, the code generator doesn't support values other than 1.
The code generator needs to check whether there are more than one incoming edge, before processing the action. This task is carried out by the code generator's Join Manager. Instead of directly generating the code of a target element, it is checked for every input pin as well as every incoming control flow whether the sum of incoming control flows and object flows of the target node is greater than one. If this is the case, a Join Manager for the target action is initialized and the generation halted. The Join Manager keeps a list of all incoming edges of the element it was created for. Every incoming edge of the target registers itself at the Join Manager before it checks whether the Join Manager allows the continuation of the generation. Only if every incoming edge registered itself at the Join Manager at least once, the Join Manager's public method isReady() returns `true` and the code generation continues.

Figure 4.4 shows a model which requires a Join Manager for its processing sequence to be correct. The generator first processes all initial nodes; in this example `ActionA` and `ActionB` since they don't have any incoming edges. After `ActionA` was processed, all succeeding nodes (`OutputPinA`, `ObjectFlow1`, `InputPinC_1`) are processed. At the processing of `InputPinC_1`, the generator calculates the number of incoming edges of the action `ActionC`. Since the the number of incoming edges is two, a *Join Manager* is initialized for `ActionC`. The node `InputPinC_1` is registered on the *Join Manager's* list and the generation is suspended since only one of the two incoming edges was registered.
The generation continues with `ActionB`, the next node of the list of initial nodes. `ActionB` and all its successors are processed until `InputPinC_2` is reached. Again, the generator checks whether the number of incoming edges of `ActionC` is greater than one. A *Join Manager*

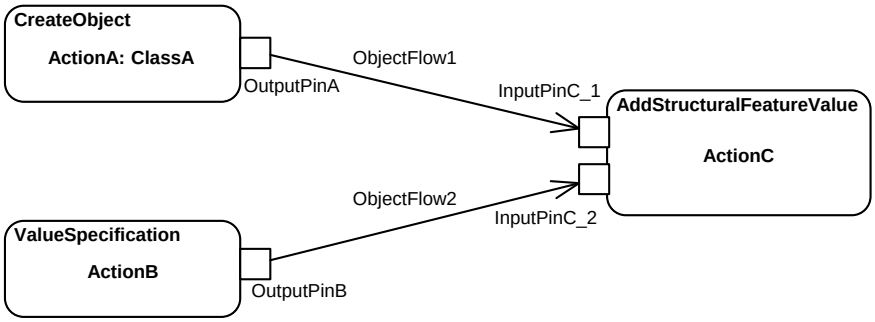

Figure 4.4: Sample fUML activity diagram illustrating the code generation for actions with multiple incoming object flows

for `ActionC` was already established and `InputPinC_2` registers itself as an incoming edge of `ActionC`. At this point, all incoming edges are satisfied and the generation can be continued with the generation of `ActionC`.

Control Nodes

Control nodes are activity nodes that are used to direct the flows between nodes. Control nodes, for which Java code is generated, are described below in a common format: An illustration of the node gives an overview of all incoming and outgoing flows. After that, a list of all properties processed by the code generator and a brief textual description of the node are given. Finally, different scenarios and their resulting Java code are used to exemplify different characteristic of the presented node.

ForkNode

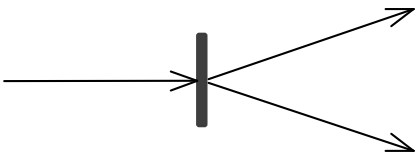

Figure 4.5: ForkNode

Processed Properties

- Incoming: ActivityEdge[0..1]
- Outgoing: ActivityEdge[0..*]

Description

Fork nodes are used to spit a flow into multiple concurrent flows. In the case of object flows,

the incoming token is provided to the target nodes of all outgoing edges by adding an entry to the internal list of variables for each outgoing edge. In order to generate concurrently executing code for the concurrent branches, Java Threads would have to be introduced. This feature is, however, not supported in the current version of the generator. Instead, the concurrent branches are processed sequentially, leading to sequentially executing code.

JoinNode

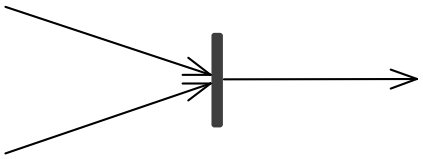

Figure 4.6: JoinNode

Processed Properties

- Incoming: ActivityEdge[0..*]
- Outgoing: ActivityEdge[0..1]

Description
Join nodes are used to combine multiple incoming flows into a single outgoing flow. The code generator therefore has to halt the generation until all incoming edges are satisfied. This behavior is provided by the code generator's JoinManager, which is also used for every action with more than one incoming edge.

DecisionNode

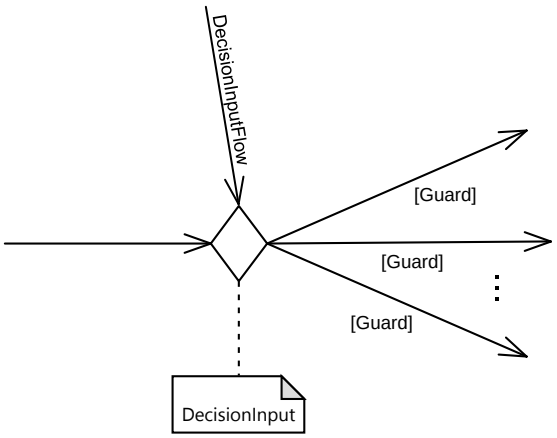

Figure 4.7: DecisionNode

Processed Properties

- DecisionInput: Behavior[0..1]
- DecisionInputFlow: ActivityEdge[0..1]
- Incoming: ActivityEdge[0..1]
- Outgoing: ActivityEdge[0..1]

Description
Decision nodes are used to direct the object flow or control flow depending on certain conditions. An optional DecisionInput, which can be an activity or opaque behavior, is used to process a value provided on the DecisionInputFlow. A decision node defines multiple outgoing edges. Every outgoing edge defines a guard which is compared to the result of the DecisionInput. If no DecisionInput is defined, the equal method is used to evaluate the incoming value against the values specified for the guards. If a DecisionInputFlow is defined without an DecisionInput behavior, the value of the the DecisionInputFlow is compared to the values of the guards while the value of the regular incoming edge is put on the outgoing edges. The generator translates this behavior into an if-statement whereby every outgoing control or object flow corresponds to an if-condition.

- **Scenario 1:** The decision node is provided with one incoming object flow. Since no decision input is defined, the value of the incoming object flow is directly compared to the value of the guards of the outgoing edges.

  Incoming: true:Boolean
  Outgoing 1: true:Boolean
  Outgoing 2: false:Boolean

```
if (new Boolean(true).equals(new Boolean(true))) {

}
else if (new Boolean(false).equals(new Boolean(false))) {

}
```

- **Scenario 2:** The decision node is provided with an incoming object flow, an decision input in form of an opaque behavior, and a decision input flow. Both incoming values are handed over to the opaque behavior. The result of the behavior is then evaluated against the values of the guards of every outgoing edge.

  DecisionInput: IntegerModulo:OpaqueBehavior
  DecisionInputFlow: 3:Integer
  Incoming: 1234:Integer
  Outgoing 1: 0:Integer
  Outgoing 2: 1:Integer

```
if ((new Integer(1234) % new Integer(3))
   == new Integer(0)) {

}
else if ((new Integer(1234) % new Integer(3))
   == new Integer(1)) {

}
```

MergeNode

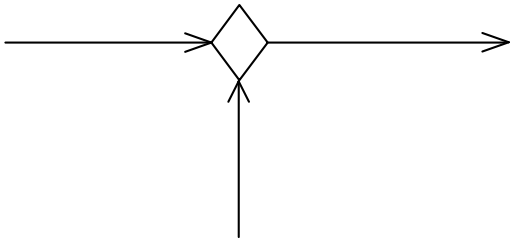

Figure 4.8: MergeNode

Processed Properties

- Incoming: ActivityEdge[0..*]
- Outgoing: ActivityEdge[0..1]

Description

Merge node are used to bring together alternative incoming flows into a single outgoing flow. The execution continues without waiting for other incoming edges. However, if the incoming flow is a result of the generation of a decision node, the generation of the outgoing nodes is suspended, because the merge node has multiple incoming edges, which causes the generated code of the following actions to be placed outside of the if-clause. Additionally, if multiple decision nodes are the only nodes that flow into one merge node, the conditions of the decision nodes are combined into one if-statement. This provides better code readability and prevents the outgoing edges of the merge node to be processes multiple times by the generator.

- **Scenario 1**

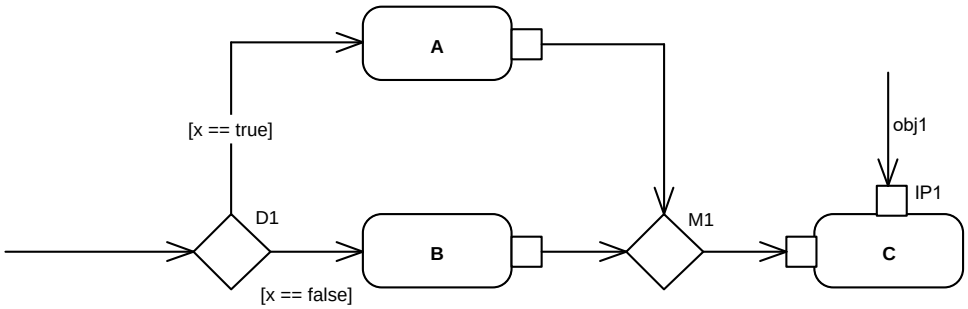

Figure 4.9: Combination of MergeNode and DecisionNode

The execution flow depicted in Figure 4.9 splits up at the DecisionNode D1. When the model is executed by the virtual machine, depending on the value of `x`, Action A or Action B is executed and thereafter Action C once. Since the code generator has to create the whole decision tree rather than one path through the tree, it has to process both outgoing edges of the DecisionNode D1.

```
if (x == true) {
   A();
   C();
}
else if (x == false) {
   B();
   C(); // <-- ERROR
}
```

Listing 4.8: Naive code generation for merge node

Listing 4.8 shows the result of the generation of the model depicted in Figure 4.9 without halting the generation at MergeNode M1. The generator first generated the code for condition `x == true`, for Action A and then processes the outgoing edges of MergeNode M1 immediately. Action C is generated for the first time, consuming the object provided at InputPin P1 and the object provided from the MergeNode M1. After the generation of Action C, its JoinManager is reset. In the next step, the condition `x == false`, Acion B is generated. The generator then tries to generated the code for Action C a second time. Since the token on InputPin P1 was already consumed, the generator can not process Action C.
To counteract this problem, the generation process is, in contrary to the behavior of the fUML virutal machine, halted at the MergeNode M1. Thus, Action C is processed only once by the generator and its code is placed outside the if-statements as shown in Listing 4.9.

```
if (x == true) {
   A();
```

```
}
else if (x == false) {
   B();
}
C();
```

Listing 4.9: Improved code generation for merge node

4.5 Mapping of fUML Actions to Java Code

This section details the code generation for the action language of fUML, i.e., the actions for which the code generator provides a translation to executable Java code. Every action is described in a common format: First, an illustration of the action gives an overview of all incoming and outgoing control flows and object flows and of the action. After that, a list of all properties processed by the code generator and a brief textual description of the action are given. Finally, different scenarios and their resulting Java code are used to exemplify different characteristic of the presented action.

Object Actions

CreateObject

Figure 4.10: CreateObject

Processed Properties

- Classifier: Classifier

Description
This action is used to create a new instance of the provided classifier. The code generator generates code that instantiates the class generated for the given classifier and declares and initializes a new variable of the provided type. The name of the newly created variable is set automatically and added to the internal list of variables to make it available in further actions. If the provided classifier is used as parameter in a ReadExtent action in any activity in the model, the variable is also added to a static collection of the class generated for the classifier.

- **Scenario 1:** The provided classifier is of the type ClassA.

  Classifier: ClassA

```
ClassA var1 = new ClassA();
```

- **Scenario 2:** The provided classifier is of the type ClassB. Additionally, an operation in model contains a ReadExtent action for the same classifier.

  Classifier: ClassB

```
ClassB var1 = new ClassB();
ClassB.addInstance(var1);
```

DestroyObject

Figure 4.11: DestroyObject

Processed Properties

- IsDestroyLinks: Boolean
- IsDestroyOwnedObjects: Boolean

Description

This action is used to destroy a provided object. Since Java does not provide any functionality to explicitly destroy objects, no code is generated for this action. However, if the provided classifier is used as parameter for a ReadExtent action, the object is removed from the collection of classifier instances. If the property IsDestroyOwnedObjects is set to true, all elements owned by the provided object are removed. This is performed by calling the clear-methods of the generated collections, if the provided object is multivalued.
If the property IsDestroyLinks is set to true, all links that participate in an association with the provided object are destroyed as well. If the property IsDestroyOwnedObjects is set to true, all objects owned by the object are destroyed as well. The functionality for these properties is not supported in the current version of the code generator.

- **Scenario 1:** The provided objects's classifier contains a ReadExtent action.

  Object: var1:ClassB

```
ClassB.removeInstance(var1);
```

ValueSpecification

Figure 4.12: ValueSpecification

Processed Properties

- ValueSpecification: ValueSpecification

Description

This action is used to specify values. For this action, the code generator creates an variable initialized with the value provided and adds it to the internal list of variables to make it accessible for further actions. In its current version, the code generator supports the specification of primitive types String, Boolean, Integer and UnlimitedNatural. The type InstanceValue is not supported.

ReadSelf

Figure 4.13: ReadSelf

Processed Properties

- *none*

Description

This action is used to read the context object of the activity containing the action. The context object of an activity is the object for which the operation associated with the activity has been called. In Java, the context of a method, in which the action is executed, is the current object instance, which can always be referenced using the keyword this. For this action, no code is generated, however, together with the action's name, the keyword this is placed in the internal list of variables to be accessible in subsequent actions.

TestIdentity

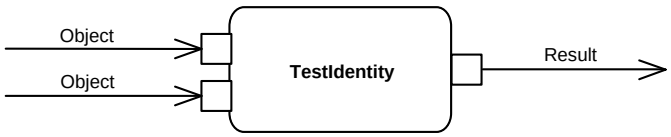

Figure 4.14: TestIdentity

Processed Properties

- *none*

Description

This action is used to test if the two provided values are identical. The values are compared with the regular equality operator if they are of primitive types. If the values are of complex types, i.e., instances of classes, the code generator calls the equals method of the first object with the second object as parameter. The result of the comparison is assigned to a new boolean variable.

- **Scenario 1:** The provided objects are of primitive types. They are compared with the equality operator. The result of the comparison is assigned to the new variable var3.

```
Boolean var3 = var1 == var2;
```

- **Scenario 2:** The provided objects are of complex types. The equal-Method of the first object var1 is called with the parameter var2, the second object. The result of the comparison is assigned to the new variable var3.

```
Boolean var3 = var1.equals(var2);
```

ReclassifyObject

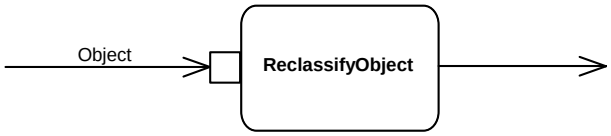

Figure 4.15: ReclassifyObject

Processed Properties

- OldClassifier: Classifier
- NewClassifier: Classifier
- IsReplaceAll: Boolean

Description
This action is used to change the type of a provided object. The generator translates this behavior into the creation of a new variable of the provided NewClassifier. The provided objects gets explicitly casted to new NewClassifier and assigned to the new variable. The parameter OldClassifier and NewClassifier are potentionally multivalued, however, since Java only casts to exactly one new type, only the first entry of the NewClassifier is considered. While the there is no restriction regarding the new type in fUML, the cast will result in a compile error in the generated code if the new type is not a sub or super type of the old type. The parameter OldClassifier and IsReplaceAll are obsolete.

- **Scenario 1:** The provided object var1 is of type ClassA and the first value of the NewClassifier collection is ClassB.

```
ClassB var2 = (ClassB)var1;
```

ReadIsClassifiedObject

Figure 4.16: ReadIsClassifiedObject

Processed Properties

- Classifier: Classifier
- IsDirect: Boolean

Description
This action is used to check whether an provided object is of a certain type. If the object is classified by the given classifier, the result is true. If the parameter IsDirect is set to equals `false`, the test returns true if the object is classified by the parameter Classifier or one of its subclasses. If IsDirect equals `true`, the provided classifier's subclasses are not considered for the test. The result of the test is provided as output of the action.

- **Scenario 1:** For the provided object var1 it is tested whether it is of type ClassA or one of its subclasses.
  Classifier: ClassA
  IsDirect: false

```
Boolean var2 = (var1 instanceof ClassA);
```

- **Scenario 2:** For the provided object var1 it is tested whether if it is of type ClassA but non of ClassA's subclasses.
  Classifier: ClassA
  IsDirect: true

```
Boolean var2 = (var1.getClass() == ClassA.getClass());
```

ReadExtent

Figure 4.17: ReadExtent

Processed Properties

- Classifier: Classifier

Description

This action is used to retrieve all instances of the provided classifier. Java does not inherently provide such a functionality. Therefore, every classifier that is used as parameter in a ReadExtent action is extended by a static collection of its own type that holds all its instances. For the collection to be consistent, it's necessary that newly created instances are added to the collection in CreateObject actions and removed in DestroyObject actions.

- **Scenario 1a:** The classifier ClassA is used as parameter in a ReadExtent action in the model. The static collection of instances and add and remove methods are added to the Java class generated for ClassA.

```
private static java.util.Collection<ClassA> instances
    = new java.util.HashSet<ClassA>();

public static java.util.Collection<ClassA> allInstances() {
   return instances;
}

public static void addInstance(ClassA instance) {
   instances.add(instance);
}

public static void removeInstance(ClassA instance) {
   instances.remove(instance);
}
```

- **Scenario 1b:** An action ReadExtent with the classifier set to ClassA is defined. All instances of ClassA are retrieved by calling its static allInstances method. The result is assigned to a new collection.
  Classifier: ClassA

```
java.util.Collection<ClassA> var1 = ClassA.allInstances();
```

Reduce

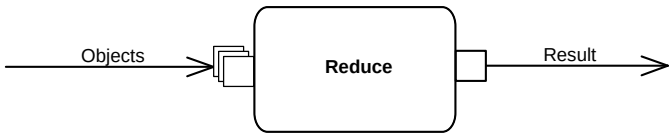

Figure 4.18: Reduce

Processed Properties

- Reducer: Behavior
- IsOrdered: Boolean

Description

This action is used to reduced a collection of objects to a single value. The specified Reducer is a behavior which is executed for every object of the incoming object. The code generator translates this behavior into a while loop that iterates over the provided input collection. In the first iteration, the Reducer behavior is called for the first two objects in the collection. In the second iteration it is called for the result of the previous call and the third element, and so on. Since the order of the method calls is determined by the collection's iterator, the parameter IsOrdered is not considered but always assumed to be true.

- **Scenario 1:** The provided collection var1 is of the type string. The Reducer method expects two parameters and returns them concatenated.
  Collection: var1: String{0..*}
  Reducer: concat(String:var1, String:var2)

```
String var2 = null;
if (!var1.isEmpty()) {
   java.util.Iterator<String> i = var1.iterator();
   var2 = i.next();

   while (i.hasNext()) {
      var2 = concat(var2, i.next());
   }
}
```

Structural Feature Actions

AddStructuralFeatureValue

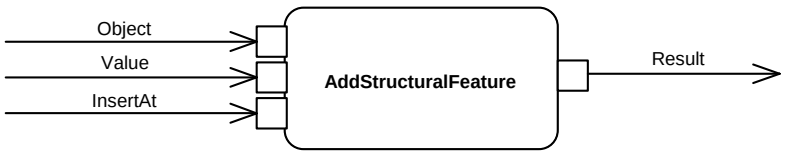

Figure 4.19: AddStructuralFeatureValue

Processed Properties

- StructuralFeature: StructuralFeature
- IsReplaceAll: Boolean

Description

This action is used to add values to structural features of objects. The code generator translates this behavior into code which modifies the values of the specified properties. The generated code depends on the multiplicity of the feature, if the feature is ordered and if the old values should be replaced beforehand.

- **Scenario 1:** In the simplest case, the structural feature is single valued and a call of the setter method for the specified property is being generated. In this scenario, neither the input value InsertAt nor the property IsReplaceAll have to be considered.

  Object: var1:ClassA
  Value: 'test1':String
  StructuralFeature: StringProperty:String of ClassA

  ```
  var1.setStringProperty("test1");
  ```

  Object: var1:ClassB
  Value: var2:ClassA
  StructuralFeature: ClassAProperty:ClassA of ClassB

  ```
  var1.setClassAProperty(var2);
  ```

- **Scenario 2:** When the structural feature is multivalued, the collection is first accessed by calling the Getter-method. If the property IsReplace is set to true, the collection is emptied before the new values is added.

  Object: var1:ClassB
  Value: var2:ClassA
  StructuralFeature: ClassCollection:ClassA[0..*]

  ```
  var1.getClassCollection().add(var2);
  ```

Object: var1:ClassB
Value: var2:ClassA
StructuralFeature: ClassAProperty:ClassA[0..*]
IsReplaceAll: true

```
var1.getClassCollection().clear();
var1.getClassCollection().add(var2);
```

- **Scenario 3:** If the structured feature is multivalued and ordered, the user can specify on which position of the collection the object shall be inserted. The collection is therefore explicitly cast to a list of the type LinkedList which provides an add-Method that inserts the provided object at the a specified position in this list. The item would be inserted in the first position, if no InsertAt parameter was provided.

  Object: var1:ClassB
  Value: var2:ClassA
  StructuralFeature: ClassCollection:ClassA[0..*], isOrdered: true
  InputAt: 3

```
((java.util.LinkedList)var1.getClassACollection())
   .add(2, var2);
```

ReadStructuralFeature

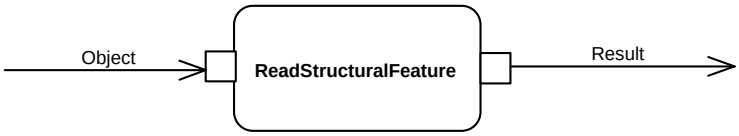

Figure 4.20: ReadStructuralFeature

Processed Properties

- StructuralFeature: StructuralFeature

Description

This action is used to read the values of a specified feature of a provided object. The generator creates a new variable of the type of the corresponding feature and assigns the value of the variable to the value defined by the input object for the defined property. The newly created variable is also added to the internal list of variables to be accessible for further actions.

- **Scenario 1:** The type of the structural feature StringProperty, and therefore the type of the newly created variable, is String. The value is obtained by calling the Getter-Method of the incoming object generated for the defined structural feature.

  Object: var1:ClassA
  StructuralFeature: StringProperty:String

```
String var2 = var1.getStringProperty();
```

- **Scenario 2:** The structural feature is multivalued, hence, the type of the new variable is a collection. The value itself is again obtained by calling the Getter method.

  Object: var1:ClassA
  StructuralFeature: ClassBCollection:ClassB[0..*]

```
java.util.Collection<ClassB> var2 = var1.getClassBCollection();
```

RemoveStructuralFeatureValue

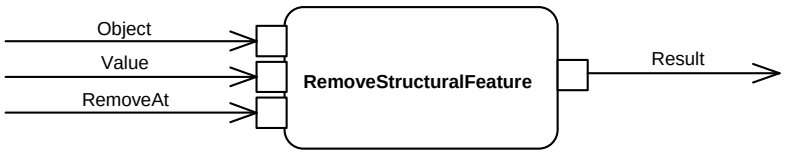

Figure 4.21: RemoveStructuralFeature

Processed Properties

- StructuralFeature: StructuralFeature
- IsRemoveDuplicates:Boolean

Description

This action is used to remove values from structural features of objects. The code generator translates this behavior into code which removes the values of the specified property. The generated code depends on the multiplicity of the structural feature, if duplicate entries shall be removed and if the incoming RemoveAt parameter is provided.

- **Scenario 1:** If the structural feature is not multivalued, the structural feature is set to a default value depending on its type. The incoming Value parameter and the RemoveAt parameter are not processed.

  Object: var1:ClassA
  StructuralFeature: StringProperty:String of ClassA

```
var1.setStringProperty(null);
```

  Object: var1:ClassA
  StructuralFeature: IntegerProperty:Integer of ClassA

```
var1.setIntegerProperty(0);
```

  Object: var1:ClassA
  StructuralFeature: BooleanProperty:Boolean of ClassA

```
var1.setBooleanProperty(false);
```

Object: var1:ClassA
StructuralFeature: RealProperty:Real of ClassA

```
var1.setFloatProperty(0.0f);
```

- **Scenario 2:** If the structural feature is multivalued, IsRemoveDuplicates is set to false and the RemoveAt parameter is provided, the collection is explicitly converted to a LinkedList, which provides an index based remove method. The incoming parameter Value is not processed.

  Object: var1:ClassA
  Value: var2:Integer
  StructuralFeature: IntegerCollection:Integer[0..*]
  RemoveAt: 2

  ```
  ((java.util.LinkedList)var1.getIntegerCollection).remove(2);
  ```

- **Scenario 3:** If the structural feature is multivalued, IsRemoveDuplicates is set to false and no RemoveAt parameter is provided, the Remove-Method of the collection with the value as parameter is called. This removes the first occurrence of the specified value from the list.

  Object: var1:ClassB
  Value: var2:ClassA
  StructuralFeature: ClassACollection:ClassA[0..*]

  ```
  var1.getClassACollection.remove(var2);
  ```

  Object: var1:ClassA
  Value: var2:Integer
  StructuralFeature: IntegerCollection:Integer[0..*]

  ```
  var1.getIntegerCollection.remove(new Integer(var2));
  ```

- **Scenario 4:** If the structural feature is multivalued and IsRemoveDuplicates is set to true, the call of the Remove-Method is nested in a loop in order to remove all occurrences of the given value. The loop exits after all occurrences are removed since the Remove-Method returns false if the provided object is not present. The RemoveAt parameter is obsolete.

  Object: var1:ClassA
  Value: var2:Integer
  StructuralFeature: IntegerCollection:Integer[0..*]
  IsRemoveDuplicates: true

  ```
  while(var1.getIntegerCollection.remove(new Integer(var2)));
  ```

ClearStructuralFeature

Figure 4.22: ClearStructuralFeature

Processed Properties

- StructuralFeature: StructuralFeature

Description

This action is used to remove all values of a specified feature of a provided object. The generated code depends on the multiplicity of the structural feature.

- **Scenario 1:** If the structural feature is not multivalued and not of a primitive type, the Setter-Method with null as parameter is called. If the structural feature is primitive typed, it is assigned the default value of the type.

  Object: var1:ClassA
  StructuralFeature: StringProperty:String

  ```
  var1.setStringProperty(null);
  ```

- **Scenario 2:** If the structural feature is multivalued, the clear method of the collection generated for the structural feature is called, which removes all elements.

  Object: var1:ClassA
  StructuralFeature: StringList:String[0..*]

  ```
  var1.getStringList().clear();
  ```

Link Actions

CreateLink

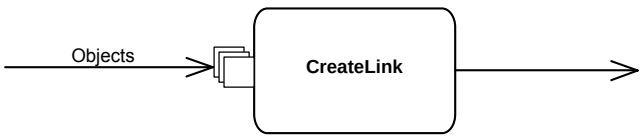

Figure 4.23: CreateLink

Processed Properties

- LinkEnd: LinkEndCreationData[0..*]
  - End: Property
  - Value: InputPin
  - InsertAt: Integer
  - IsReplaceAll: Boolean

Description
This action is used to create a new link between a set of provided input object according to the specified link creation data. The code generator translates this behavior into assigning the properties corresponding to the navigable end of an association to the association value. While the number of potential incoming link ends is unlimited, only binary associations are supported. The generated code depends on the multiplicity of the association ends, if they are navigable and ordered, and if existing links shall be removed. If the association end is navigable and single valued, its value is assigned by calling the setter method of the corresponding property. If an association end is navigable and multivalued, the reference is established by calling the getter method of the generated property and then add the association end value to the collection.

- **Scenario 1:** As depicted in Figure 4.24, the provided link ends are part of an association. Both association ends ends are navigable and one of the association ends is multivalued.

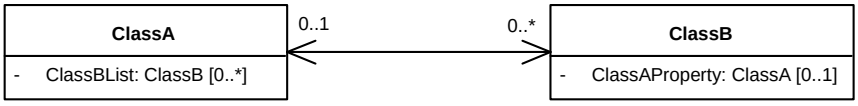

Figure 4.24: Class diagram for Scenario 1 of CreateLink

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: true
Value: var2:ClassA

Link End 2
End: ClassBList:ClassB[0..*], navigable: true
Value: var2:ClassB

```
var1.getClassBList.add(var2);
var2.setClassAProperty(var1);
```

- **Scenario 2:** As depicted in Figure 4.25, the provided link ends are part of an association, but only one of the association ends is navigable. The generator creates a link between the navigable end and the value.

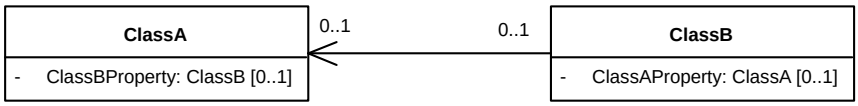

Figure 4.25: Class diagram for Scenario 2 of CreateLink

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: true
Value: var1:ClassA

Link End 2
End: ClassBProperty:ClassB[0..1], navigable: false
Value: var2:ClassB

```
var1.setClassBProperty(var2);
```

- **Scenario 3a:** As depicted in Figure 4.26, the provided link ends are part of an association but only one of them is navigable. One of the association ends is multivalued and ordered. The position, at where the link shall be added to the collection, is provided by the InsertAt property.

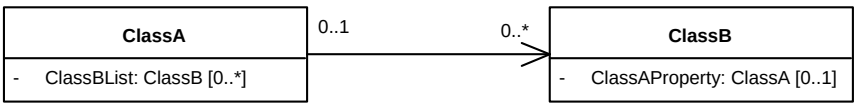

Figure 4.26: Class diagram for Scenario 3a and 3b of CreateLink

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: false
Value: var1:ClassA

Link End 2
End: ClassBList:ClassB[0..*], navigable: true
Value: var1
InsertAt: 3

```
((java.util.LinkedList)var1.getClassBList()).add(2, var2);
```

- **Scenario 3a:** As depicted in Figure 4.26, the provided link ends are part of an association but only one of them is navigable. One of the association ends is multivalued but not ordered. The property ReplaceAll is set to true.

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: false
Value: var1:ClassA

Link End 2

End: ClassBList:ClassB[0..*], navigable: true
Value: var1
IsReplaceAll: true

```
var1.getClassBList.clear();
var1.getClassBList().add(var2);
```

ReadLink

Figure 4.27: ReadLink

Processed Properties

- LinkEnd: LinkEndData[0..*]
  - End: Property
  - Value: InputPin

Description

This action is used to read the value at one end of a link given the other linked objects provided as input. While the number of incoming objects is potentially unlimited, the code generator only supports binary associations, i.e. one object has to be provided as input and the other linked object is provided as output of a ReadLink action. To specify which end of the link shall be returned, one of the link ends is defined as target end. The code generator creates a new variable depending on the type and multiplicity of the target end and adds the variable to the internal variable list.

- **Scenario 1:** Two link ends are provided but only for Link End 1, a value is provided. Therefore, the other end of the association ClassBProperty is returned and assigned to the new variable var3.

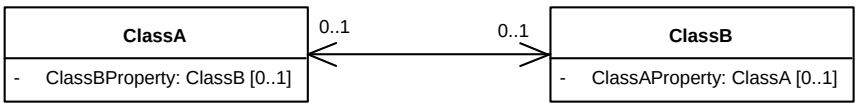

Figure 4.28: Class diagram for Scenario 1 of ReadLink

Link End 1

End: ClassAProperty:ClassA[0..1]
Value: var1:ClassA

Link End 2

End: ClassBProperty:ClassB[0..1]
Value: var2:ClassB

```
ClassB var3 = var1.getClassBProperty();
```

- **Scenario 2:** Again, two link ends are provided and only for Link End 1, a value input pin is assigned. The other end of the association ClassBList is multivalued. Therefore, the new variable var3 is declared as a collection.

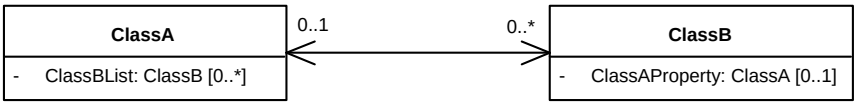

Figure 4.29: Class diagram for Scenario 2 of ReadLink

Link End 1

End: ClassAProperty:ClassA[0..1]
Value: var1:ClassA

Link End 2

End: ClassBList:ClassB[0..*]
Value: var1

```
java.util.Collection<ClassB> var3 = var1.getClassBList();
```

DestroyLink

Figure 4.30: DestroyLink

Processed Properties

- LinkEnd: LinkEndDestructionData[0..*]
  - End: Property

- Value: InputPin
- DestroyAt: Integer
- IsDestroyDuplicates: Boolean

Description
This action is used to destroy links between the provided input objects according to the specified link end destruction data. While the number of incoming objects is potentially unlimited, the code code generator only supports binary associations, i.e. two link ends. The generated code depends on the multiplicity of the member ends. If the end is multivalued, the generation additionally depends on whether the link is ordered and whether duplicate vales shall be removed.

- **Scenario 1:** The two provided link ends are part of an association, but only one of the association ends is navigable. Since the link ends are not multivalued, the link is destroyed by assigning null to the navigable link end.

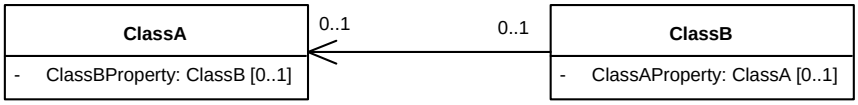

Figure 4.31: Class diagram for Scenario 1 of DestroyLink

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: true
Value: var1:ClassA

Link End 2
End: ClassBProperty:ClassB[0..1], navigable: false
Value: var2:ClassB

```
var2.setClassAProperty(null);
```

- **Scenario 2:** The two provided link ends are part of an association. Both association ends are navigable and one of the association ends is multivalued. The properties DestroyAt and IsDestroyDuplicates are not provided.

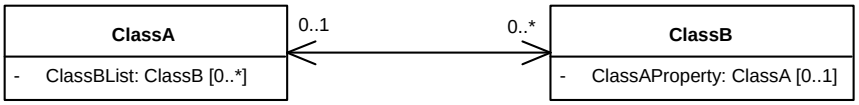

Figure 4.32: Class diagram for Scenarios 2, 3 and 4 of DestroyLink

Link End 1
End: ClassAProperty:ClassA[0..1], navigable: true
Value: var1:ClassA

Link End 2

End: ClassBList:ClassB[0..*], navigable: true
Value: var2:ClassB

```
var1.getClassBList.remove(var2);
var2.setClassAProperty(null);
```

- **Scenario 3:** The two provided link ends are part of an association. Both association ends are navigable and one of the association ends is multivalued. Additionally, the property IsDestroyDuplicates on the multivalued link end is set to true;

Link End 1

End: ClassAProperty:ClassA[0..1], navigable: true
Value: var1:ClassA

Link End 2

End: ClassBList:ClassB[0..*], navigable: true
Value: ClassB: var2
IsDestroyDuplicates: true

```
while(var1.getClassBlist().contains(var2)) {
   var1.getClassBList.remove(var2);
}
var2.setClassAProperty(null);
```

- **Scenario 4:** Two link ends are provided. One of the association ends is multivalued and ordered. The position, at which the link shall be destroyed, is provided by the DestroyAt property. The property IsDestroyDuplicates is not provided.

Link End 1

End: ClassAProperty:ClassA[0..1], navigable: true
Value: var1:ClassA

Link End 2

End: ClassBList:ClassB[0..*], navigable: true
Value: var2:ClassB
DestroyAt: 3

```
if (((java.util.LinkedList)var1.getClassBList())
   .get(3).equals(var2)) {
   ((java.util.LinkedList)var1.getClassBList()).remove(3);
}
var2.setClassAProperty(null);
```

Communication Actions

CallBehavior

Figure 4.33: CallBehavior

Processed Properties

- Behavior: Behavior

Description

This action is used to directly invoke a specified behavior. If the behavior is an activity, a call to the operation associated with the activity is generated. If the specified behavior is of the type OpaqueBehavior, code implementing the behavior of this opaque behavior is generated. While the number of outgoing values, according to the fUML specification, is unlimited, only zero or one return values are supported.

- **Scenario 1:** The activity validate is called with one argument pin.

  Behavior: Activity of validate:Operation
  Argument: pin:Type

  ```
  validate(pin);
  ```

- **Scenario 2:** The opaque behavior IntegerTimes is called with the arguments 4 and 3. The result of the behavior is assigned to a newly created variable.

  Behavior: IntegerTimes:OpaqueBehavior
  Argument 1: 3:Integer
  Argument 2: 4:Integer

  ```
  Integer var1 = new Integer(3) * new Integer(4);
  ```

  OpaqueBehaviors are behaviors for which no semantics are defined in the UML standard. They are used to perform primitive operations on primitive data types defined in the fUML specification [47] and expect sets of input parameter values and output parameter values. The execution of OpaqueBehaviors depends solely on the input values and does not interfere with structural features or link values of their containing elements. Their behavior is, per definition, completely self contained. The code generator maps the name of an OpaqueBehavior to a specific code template. The template contains meta-markers which are replaced with the passed parameter values. The behavior is then evaluated during runtime.

The following list contains all OpaqueBehaviors supported by the current version of the code generator:

- IntegerLess
- IntegerLessOrEqual
- IntegerGreater
- IntegerGreaterOrEqual
- IntegerPlus
- IntegerMinus
- IntegerTimes
- IntegerModule
- BooleanAnd
- BooleanOr

CallOperation

Figure 4.34: CallOperation

Processed Properties

- Operation Operation

Description

This action is used to invoke a specified operation for the provided target object. The code generator translates this behavior into a method call on the specified object. The generated code depends on the provided arguments and the return type of the specified operation. An operation is limited to one return result, while the number of outgoing values, according to the fUML specification, is unlimited. The code generator, however, only supports an unlimited number of incoming arguments and zero or one return values.

- **Scenario 1:** The operation toUpper is called with the argument value on the object var1. The result of the method is assigned to the newly created variable var2.

  Target: var1:ClassA
  Operation: toUpper:String
  Argument: value:String

```
String var2 = var1.toUpper(value);
```

- **Scenario 2:** The operation getZipCodes is called on the object var1 without arguments. Since the return type of the operation is multivalued, the result of the method is assigned to the newly created string collection var2.

  Target: var1:ClassA
  Operation: getZipCodes:String[0..*]

```
java.util.Collection<String> var2 = var1.getZipCodes();
```

Structured Activity Nodes

ExpansionRegion

Figure 4.35: ExpansionRegion

Processed Properties

- Input Element: ExpansionNode[0..*]
- Output Element: ExpansionNode[0..*]

Description

ExpansionRegions are actions that processes an unlimited number of incoming and outgoing collections provided by Expansion Nodes. An expansion node is either defined as input, output or input and output node of an expansion region. The region is executed multiple times corresponding to the number of objects in the first input expansion node. For every output expansion node, a new collection is initialized. Based on the length of the first incoming expansion node, the code generator creates a for loop in which the objects of the input collections are iterated and provided to the elements defined in the body of the expansion region successively. After the code for all nodes within the expansion region is generated, all objects collected as output for the output expansion nodes are added to the result collections and this way provided to be used outside the region.

- **Scenario 1:** The expansion defines two incoming expansion nodes and one outgoing expansion node. For the outgoing expansion node, a new variable is created, added to the internal VARS list, and assigned to a new collection. The incoming nodes are accessed inside the iteration and thus assigned to new variables. By that, the elements defined within the expansion region are provided with the individual elements of the incoming collections. At this point, the code generator proceeds with the generation of the code for the nodes defined within the region. After that, all objects associated with an outgoing

expansion node are added to their corresponding collection, making them accessible for succeeding actions.

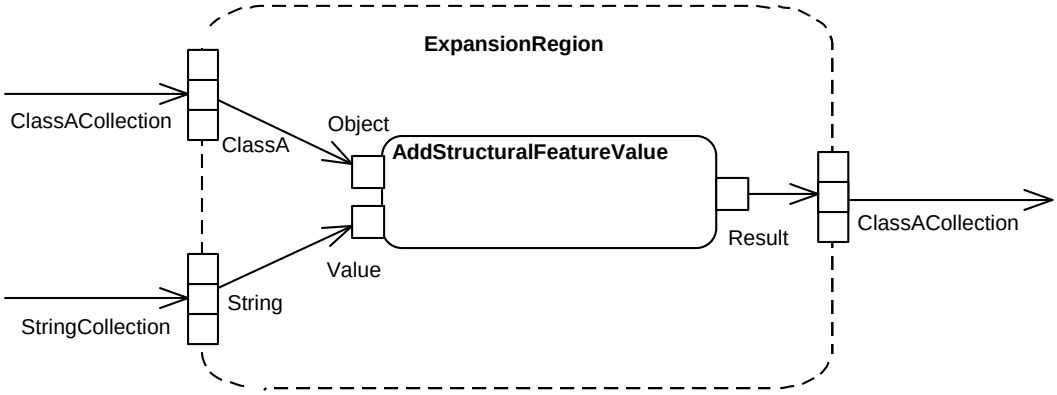

Figure 4.36: Expansion Region for Scenario 1

Expansion Node 1 (in): var4:ClassA[0..*]
Expansion Node 2 (in): var5:String[0..*]
Expansion Node 3 (out): var6:ClassA[0..*]

```
java.util.Collection<ClassA> var6 =
   new java.util.ArrayList<ClassA>();

for(int i = 0; i < var4.size(); i++) {
   ClassA ER1_var1 = ((java.util.ArrayList<ClassA>)var4).get(i);
   String ER1_var2 = ((java.util.ArrayList<String>)var5).get(i);

   ER1_var1.setName(ER1_var2);
   var6.add(ER1_var1);
}
```

4.6 Limitations

Limited Support

The following list contains fUML actions and other fUML elements that are processed by the code generator but only partly supported or have certain limitations in regard to their specification.

- Class Diagrams
  - **Sub and super classes**
    Generalizations between classes are not supported.
  - **Visibility**
    The visibility types *Protected* and *Package* of properties are not supported.
  - **Parameter direction**
    Operations can contain unlimited *in* parameters and zero or one *return* parameter while the parameter directions *out* and *in out* are not supported.
  - **Associations**
    Only binary associations, i.e. associations with exactly two member ends, are supported.
- Activity Diagrams
  - **Token flow**
    The UML specifications allows unlimited tokens to be required at an node's input pin for the node to be executed. The code generator does not support other values than 1.
  - Object Actions
    * **DestroyObject**
      The property IsDestroyLinks, which, if set to true, deletes all objects that participate in an associations with the input object, is not supported.
      The property IsDestroyOwnedObject, which, if set to true, destroys all objects that are owned by the deleted object, is not supported.
    * **ValueSpecification**
      The type InstanceValue is not supported.
    * **ReclassifyObject**
      fUML allows the unrestricted reclassification of objects. This behavior is not supported by the code generator since Java only allows objects to be converted to sub or super types of their current type.
  - Link Actions
    All supported link actions (**CreateLink**, **ReadLink** and **DestroyLink**) are limited to binary associations.

  - Communication Actions
    CallBehavior actions and **CallOperation** actions may contain unlimited *in* parameters and zero or one *return* parameters. Parameters with the direction *out* and *in out* are not supported.

Unsupported Actions

The following list contains fUML actions that are not supported by the current version of the code generator.

- Communction Actions
  - **StartClassifierBehavior**
    This action accepts an object and starts a behavior defined as *Classifier Behavior* of the classifier of the object. To provide such functionality, Java constructors could be used. However, similar functionality can be achieved with *CallBehavior* or *CallOperation* actions.
  - **StartObjectBehavior**
    This action is used to start the behavior of a provided object. If the provided object is an instance of a behavior, this behavior is executed. Java does not support function objects, however, interfaces with anonymous inner classes that implement the interface or the lambda expression features in Java 8 could provide such behavior. If the provided object is not an instance of a behavior, the classifier behavior of the provided object's type is started.
  - **SendSignal**
    This action is used to create an instance of a signal and send it to a target object asynchronously. A possibility to provide such functionality would be the generation of an observer pattern. To provide a truly asynchronous behavior in Java, multiple threads would have to be set up.
  - **AcceptEvent**
    This action waits for the occurrence of specific events. *Triggers* are used specify the type of accepted events. To provide such functionality, event listener in form of Java ActionListeners could be generated.
- Link Actions
  - **ClearAssociation**
    This action accepts an object and destroys all links of the association in which the object participates. To provide this functionality, the generator would have to keep track of all references between objects and destroy them by setting the referenced objects to `null`.

- Structured Activity Nodes
  - **ConditionalNode**
    This node consists of clauses, whereby every clause consists of a test section and a body section. If a test section yields true, its corresponding body section is executed. However, similar functionality can be achieved by the use of *DecisionNodes*.
  - **LoopNode**
    This node consists of three sections; a setup section is used to initializes values or perform computations for the loop, a test section is used to compute a Boolean value and a body section contains the repetitive computation and is executed as long as the test section yields true. To provide such a functionality, a compound Java `do-while-statement` could be used.

CHAPTER 5

Code Verification

This chapter describes the process of verifying the correctness of code generated with the developed fUML code generator. After an overview of the requirements on the developed code verification component is given, the execution event tracing (EET) metamodel that is used to capture runtime information about the execution of an fUML model and the execution of the Java code generated for the fUML model is presented. After that, extensions of the fUML virtual machine that provide events during execution of a model are presented and it is shown how the events captured by the fUML virtual machine are transformed into a model conforming to the EET metamodel. Then, an overview of AspectJ is given and it is shown how AspectJ is used for monitoring and tracing the execution of Java code and the transformation of AspectJ events into the EET metamodel is given. Finally, the method for comparing two instances of the EET metamodels is presented. The developed software artefacts are available online. Further information about the public repository are provided in Appendix A.

5.1 Overview

After an fUML model was transformed into Java code, the Java code's correctness must be ensured. This task is carried out by the implementation of a code verification approach. To verify the correctness of the generation of the static model parts, it must be verified that all structural features of the fUML model, such as classes, properties, etc. are reflected in the generated code. For the dynamic part of the model to be correct, is must be verified that the execution of the generated code behaves equivalent to the execution of the model carried out by the fUML virtual machine. To do so, both executions have to be monitored closely; every creation and modification of an object is captured and stored in a comparable form, which is then checked for equality. The result of the comparison contains the differences of both executions; if no differences are found, the execution of the generated code is considered to be equivalent to the execution carried out by the fUML virtual machine.

In the following, we give a detailed overview of the developed code verification approach. The verification process is illustrated in Figure 5.1. An fUML model (1) is handed over to the

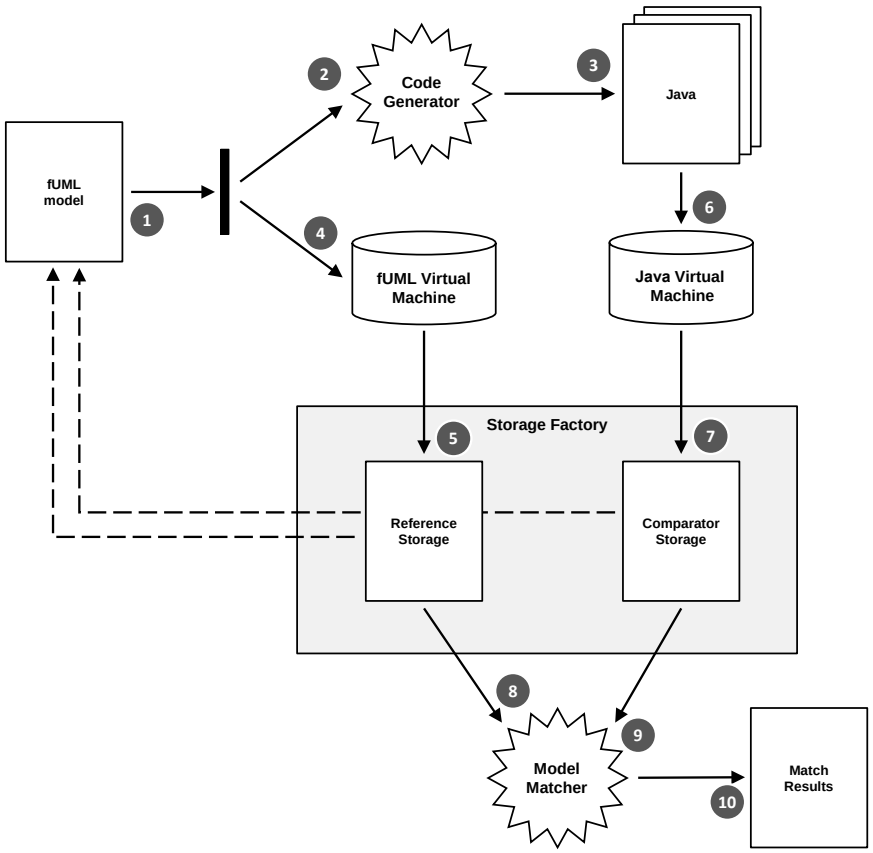

Figure 5.1: Overview of the code verification process

code generator (2) which transforms the model into Java code (3) as described in Chapter 4. The input fUML model is then handed over to the fUML virtual machine and executed (4). The event model, presented in Section 5.3, of the fUML virtual machine is used to capture all relevant events that occur during the execution process in the so-called reference storage (5), which is a model conforming to the execution event tracing metamodel that will be explained in Section 5.2. The Java code, which was generated from the same input fUML model, is then executed (6). Every object creation and object modification is captured through AspectJ point cuts as described in Section 5.4. The captured events are also stored in a model conforming to the execution event tracing metamodel called comparator storage (7). This is decribed in Section 5.4. After both executions are completed, the two instances of the execution event tracing metamodel are handed over to the *Model Matcher* (8, 9) where every element of the *Reference Storage* and the *Comparator Storage* are matched using the algorithm described in Section 5.5. This matching is done with EMFCompare [5]. The result of the matching process is stored in an output model. If a match is found for every element, the code generation is considered to be correct, i.e., the Java code is considered to be equivalent to the original fUML model.

5.2 Execution Event Tracing Metamodel

The Execution Event Tracing metamodel (EET), depicted in Figure 5.2, is used to capture and later on compare runtime information about the executions of fUML models and Java code generated for them. The metamodel's root element Storage contains two collections: the collection `ObjectStore`, consisting of `ObjectStoreItems`, which are elements created during the execution, and the collection `EventStore`, consisting of `ExtensionalValueEvents` and `FeatureValueEvents`, which monitor every creation of objects and modifications of objects that are stored in the collection `ObjectStore`. `ObjectStoreItems` point to `ExtensionalValues` with an additional property hashValue, which is necessary to identify captured objects during the execution. The remaining referenced model elements are elements contained by the fUML execution model as well as the UML metamodel: `ExtensionalValues` are objects or links created during the execution of an fUML model or the generated code. `FeatureValues` represent the values of the objects' properties. Every property of an object is represented by one `FeatureValue`. The `feature` reference of a FeatureValue references the associated UML property and the `values` containment reference contains the feature's values. If the `feature`, however, is a member end of an association and not a property owned by a class, the feature values represent the references to the linked objects.

5.3 Tracing the Execution of fUML Models

The fUML virtual machine executes a specified activity defined within a provided fUML model. After the activity is executed, the output parameter values are provided as result. With the fUML virtual machine and its implementation of the standardized behavioral semantics, UML was enhanced to an executable modeling language. As described in Section 3.2, the biggest potentials for executable models lies in their ability to be easily tested, and to be analysed and debugged at runtime. Important features like the observation, analysis and control of the execution of a model, are not supported by the fUML virtual machine. To overcome this limitation and form a basis for model analyses, an event model, a trace model, and a command API were introduced in the fUML virtual machine [31]. In this work, we made use of the introduced event model to capture the runtime information necessary to verify the correctness of the code generation for fUML models.

The essence of the event model is to capture every change in the runtime state of an executing fUML model and trigger a corresponding event. An excerpt of the event metamodel is depicted in Figure 5.3. Events of the type `ExtensionalValueEvent` capture the creation, modification an destruction of objects, while `FeatureValueEvents` notify about the modification of the feature values of an extensional value (i.e. the modification of the property values of an object). By implementing the `ExecutionEventListener` interface, the developed code verification component is able to receive all events which occur during the execution. Every captured `ExtensionalValueEvent` and `FeatureValueEvent` is passed to the `StorageFactory` and transformed to the EET metamodel.

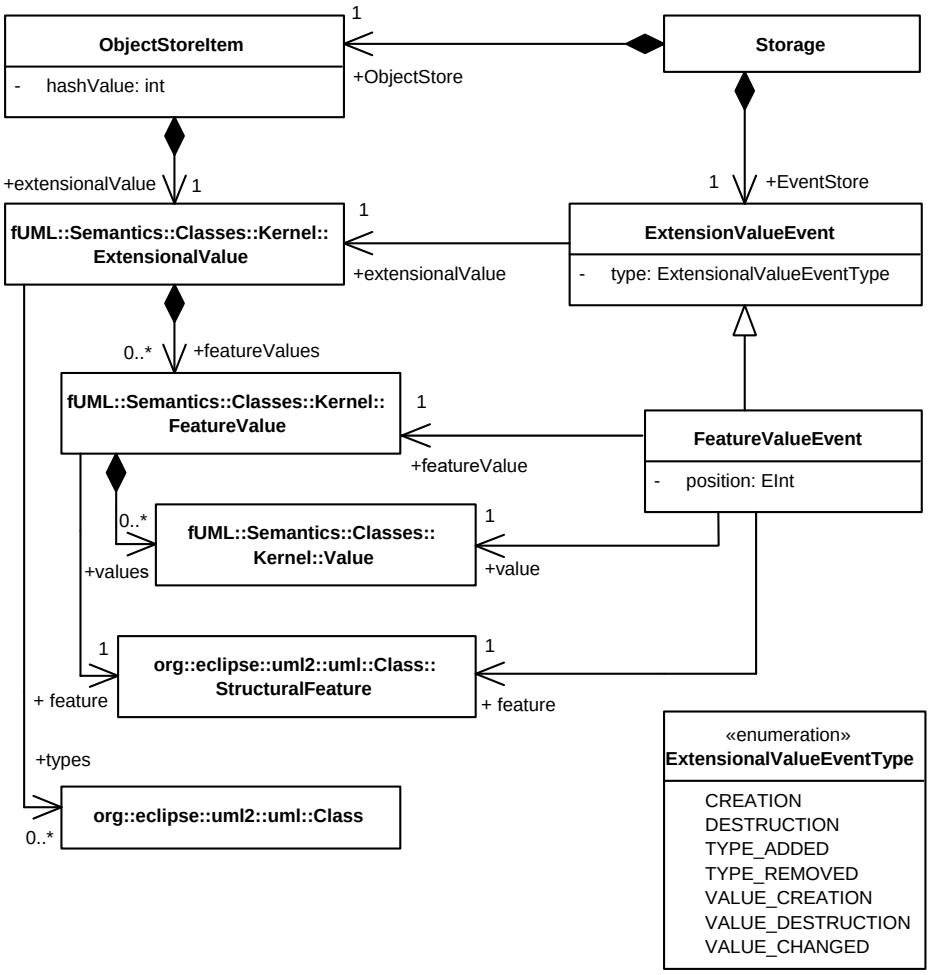

Figure 5.2: Execution Event Tracing metamodel

Creation of Event Storages

In order to compare the information captured for the execution of an fUML model with the information captured for the execution of the code generated for the fUML model, the gathered information is transformed into the EET metamodel depicted in Figure 5.2. This task is carried out by the `StorageFactory`, which contains two instances of the metamodel's root element `Storage`; one for elements captured by the events triggered by the fUML virtual machine, the `ReferenceStorage`, and one captured by AspectJ point cuts, the `ComparatorStorage`. The `StorageFactory` provides a set of public methods to transform the captured events into new objects of the EET metamodel. The `StorageFactory`, which is implemented as a singleton, is initialized with a reference to the fUML model resource file. Before any events are captured, every UML class in the resource is stored with its name as key, and every UML property is stored with its name and its parental class name as key. This allows the captured events to establish the references to the fUML model elements during the event storing process.

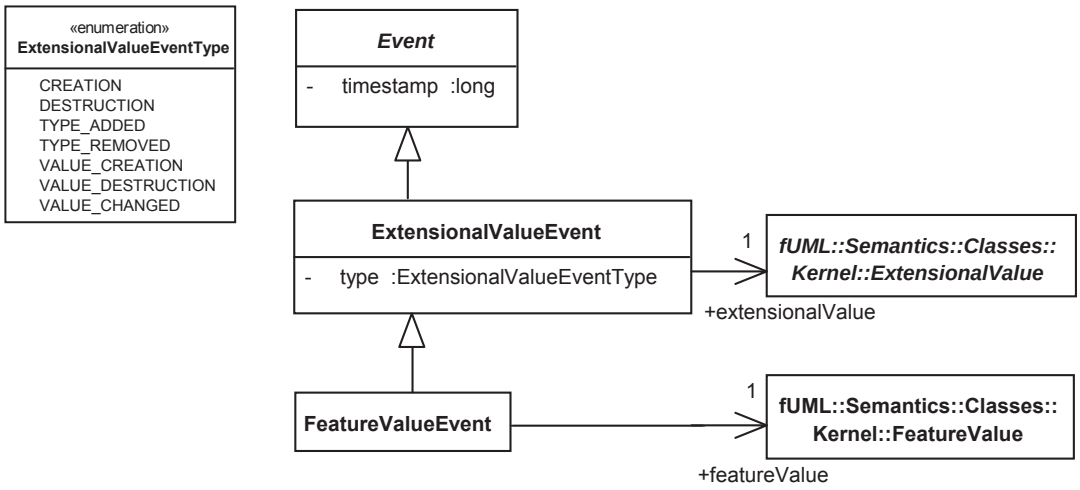

Figure 5.3: fUML event metamodel for fUML [31]

After the fUML resource is processed, the `StorageFactory` is ready to process events issued by the fUML virtual machine as well as from the AspectJ point cuts as described in Section 5.4.

Transformation of Trace Events

The events captured by the fUML virtual machine are added to the `ReferenceStorage` rather straightforwardly; in the case of an `ExtensionalValueEvent` being captured, a new `ObjectStoreItem` element with a reference to the `ExtensionalValue` is added to the `ObjectStore`. Every `FeatureValue` of the `ExtensionalValue` referenced by the `ExtensionalValueEvent` is added to the `ExtensionalValue` of the new `ObjectStoreItem` and linked with the corresponding property from the fUML model.

Properties owned by Associations

The concept of associations as a relationship between objects only exists in UML class diagrams but not in Java, and therefore as a valid entity in the static part of fUML models, but not in the Java code generated for the fUML model. To overcome this mismatch, associations in fUML models are resolved into properties as described in Section 4.3 in the code generation process. This needs to be taken under consideration for the events captured from the execution of the model by the fUML virtual machine: when an `ExtensionalValueEvent` is captured, the `ExtensionalValue`'s type only contains the properties that it owns as defined in the class diagram. However, as shown in Figure 5.4, properties can not only be owned by classes but also by associations. If the model contains associations with owned member ends (i.e. it ownes and defines the associated property), the Java class may contain additional properties in the generated code, that are not contained in the captured `ExtensionalValue`. To ensure that the `ExtensionalValue`'s type captured by the fUML virtual machine contains all properties

as if it was translated into Java code, all associations are processed and missing properties are added manually to the `ExtensionalValue` when it is initially added to the `ObjectStore`.

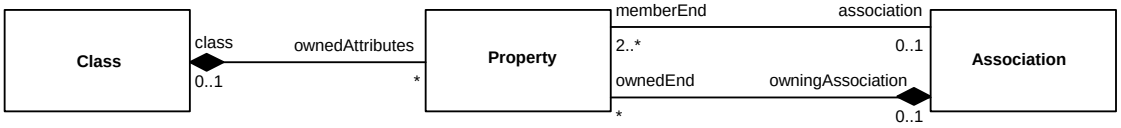

Figure 5.4: Metamodel for UML classes, properties and associations

Resolving Linked Objects

The modification of member end values of an association during execution of the model with the fUML virtual machine caused by e.g. a `CreateLinkAction`, will also result in an `ExtensionalValueEvent`, whereby the `ExtensionalValue` of the event is the `Link` object itself. During code generation, the `CreateLinkAction` is resolved into the modification of all navigable properties as described in Section 4.5, which then result in a set of `FeatureValueEvents` captured by AspectJ during the generated code's execution. In order to provide comparability between the two captured sets of events, `ExtensionalValueEvents` with `ExtensionalValues` of the type `Link` are resolved into `FeatureValueEvents` for each navigable member end of the association.

Example. To illustrate this functionality, we consider the execution of the activity depicted in Figure 5.5. The two classes Student and Lecture are connected with an association. The two association ends student and favouriteLecture are not owned by the classes but by the association. The code generator, as described in Section 4.3, resolves the association and adds the property favouriteLecture to the generated Java class Student. For the other association end student, no property is generated since it is not navigable. The generated code for the depicted activity is shown in Listing 5.1.
When executed, AspectJ captures three events: the creation of two objects of the type Student and Lecture and the modification of the feature favouriteLecture on the object var1. This results in

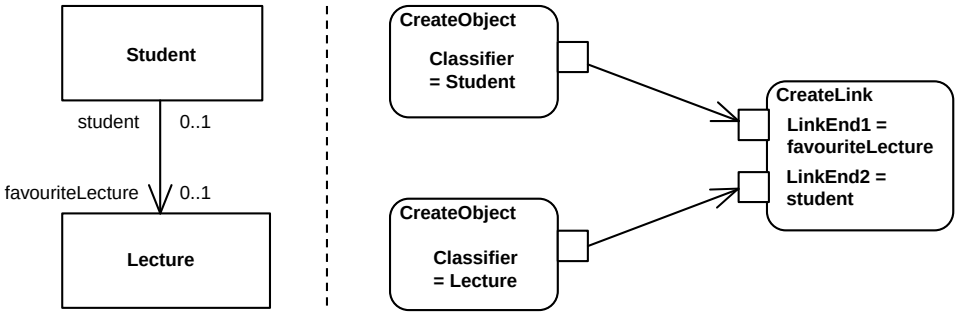

Figure 5.5: Sample fUML class and activity diagrams illustrating the creation of links

the creation of two ExtensionalValueEvents and one FeatureValueEvent, which are added to the ObjectStore. It is important to note, that the created ExtensionalValue for the Student object contains a FeatureValue for the property favouriteLecture, since it was added to the Java class by the code generator.
When executed by the fUML virutal machine, the ExtensionalValue of the captured ExtensionalValueEvent caused by the execution of the CreateObjectAction does not contain the FeatureValue favouriteLecture, since it is not owned by the class but the association. The FeatureValue owned by the association is therefore added to the ExtensionalValue. The execution of the CreateLinkAction by the fUML virtual machine results in an ExtensionalValueEvent with the link as ExtensionalValue. Such event, since associations in this form do not exist in Java code, can impossibly emit from the execution of generated code. To provide comparability, the ExtensionalValueEvent is therefore transformed into an FeatureValueEvent that is equal to the FeatureValueEvent captured from the execution of the generated code.

```
Student var1 = new Student();
Lecture var2 = new Lecture();
var1.setFavouriteLecture(var2);
```

Listing 5.1: Code generation result for the activity shown in Figure 5.5

5.4 Tracing the Execution of the Generated Java Code

To monitor the execution of Java code generated for an fUML model, AspectJ[1] is used. AspectJ is an programming language that extends Java to provide the application of the aspect-orientated programming (AOP) paradigm. AOP addresses crosscutting requirements that span over multiple classes and by that oppose the hierarchically modularized approach of object-orientated programming (OOP).
OOP is based on a few essential techniques; *Ecapsulation* describes the idea of making objects responsible for a certain objective. OOP languages structure programs into classes, which include properties that hold data, and methods that modify the data. By instantiating a class, a new self-contained object is created. *Inheritance* allows objects to inherit features from other objects by defining subclasses which share characteristics from parental classes. *Polymorphism* describes the technique of objects sharing a common interface but being of different types at runtime. All these design features are targeted at providing functionality through hierarchically modularized programs. However, particular concerns may result in situations that do not align well with the modular OOP approach, i.e. they cross-cut the system's modularization.
AspectJ addresses this demand providing a set of AOP concepts; *Pointcuts* allow cross-cutting the primary modularization. They pick out *Join Points*, which are well-defined points in the execution of a program. *Advices* are used to combine a pointcut and a block of code, which can be executed before, after or around specified join points. Different kinds of point cuts (method call,

[1]The AspectJ Documentation and Resources are available at http://www.eclipse.org/aspectj/doc/released/index.html, accessed 18-02-1017

field set, object initialization, ..) are use to point to certain events [20]. These different concepts are then combined into *Aspects* to serve a specific AOP purpose.
In the scope of the code verification, the AOP purpose is to monitor the creation and modification of every object during the execution of the generated Java code. When a specified point cut occurs during the execution, a method of the `StorageFactory` with values of the execution context is called to add a new event to the `StorageFactory`. To monitor the execution of the generated code, the following point cuts where established:

1. **ObjectInitialized**
   The initialisation of every type of object is covered by this point cut. A reference to the created object is passed to the `StorageFactory`. The keyword `new(..)` is used to filter out the initialisation event of the hosting class of the method, whose execution is being monitored.

```
pointcut ObjectInitialized(): initialization (new(..));
```

2. **SimpleObjectModified**
   This point cut fires whenever the value of a primitive property of an object was modified. The method SimpleType() returns true if the type of the affected property is of one of the eight primitive Java types. A reference to the object and the name of the modified property are passed to the `StorageFactory`, where the value is obtained via reflection.

```
pointcut SimpleObjectModified(): set(* *) && SimpleType();
```

3. **ObjectModified**
   The modification of every object field, that is not of a primitive type and not of a collection type is monitored by this point cut. A reference to the object, the name of the affected property, as well as a reference to the object which is the new value of the modified property are passed to the `StorageFactory`.

```
pointcut ObjectModiefied():
   set(* *) && !SimpleType() && !CollectionType()
```

4. **ObjectAddedToCollection**
   Every time an object is added to a collection, this point cut fires. The object representing the collection as well as the reference to the added object are passed to the `StorageFactory`. If the type of the collection is non-unique, an additional parameter, the index where the new object shall be inserted, is passed to the `StorageFactory`.

```
pointcut ObjectAddedToCollection():
      call(* java.util.Collection.add*(..))
```

5. **ObjectRemovedFromCollection**
   Every time an object is removed from a collection, this point cut fires. The object representing the collection as well as the reference to the removed object are passed to the `StorageFactory`. If the type of the collection is non-unique, the index from where the object was removed, is passed to the `StorageFactory`.

```
pointcut ObjectRemovedFromCollection():
      (call(* java.util.Collection.remove*(..))
```

Transformation of AspectJ Events

During the execution of the generated code, every creation and modification of objects is captured by the described AspectJ point cuts. Along with context data, the affected object is passed to the `StorageFactory`, where, depending on the type of the event, new elements are added to the `ObjectStore` and `EventStore` of the `ComparatorStorage`.

The creation of a new object, captured by the AspectJ pointcut ObjectInitialized, results in the creation of a new `ObjectStoreItem` as well as new `ExtensionalValueEvent` by passing the object to the `StorageFactory`. Since the name of the created object is not derivable from the creation context, its hash value is stored in the new `ObjectStoreItem` to be able to identify and reference it later on. In the next step, a new `ExtensionalValue` is created. The type of the `ExtensionalValue` is set by looking up the UML class from the fUML model with the class name of the passed object. The object is further examined by iterating over all its declared fields in the next step. For every field of the object, a new `FeatureValue` is added to the `ExtensionalValue`. The corresponding UML property is looked up in the fUML model from the properties list and referenced by the `feature` reference.

After the new item was added to the `ObjectStore`, a new `ExtensionalValueEvent` with the type CREATION and a reference to the newly created `ExtensionalValue` are added to the `EventStore`.

When an object is modified, which is detected by the poitcuts SimpleObjectModified, ObjectModified and ObjectAddedToCollection, a new `FeatureValueEvent` is added to the `EventStore`. The `ExtensionalValue` corresponds to the modified object, which is represented by an `ExtensionalValue` created before and looked up with its internal hash code. The modified feature is set to the corresponding UML property. If the value is of a primitive type, a new `PrimitiveValue` is created, casted to the correct data type (Boolean, Integer, String or UnlimitedNaturalValue) and set to the passed value using reflection, since the declared fields of the object are not known at compile time. If the value, however, is not of a primitive type, a new `Referece` with the referent set to the `ExtensionalValue` of the referenced object is created. The type of the `FeatureValueEvent` is set to `VALUE_ADDED`.

5.5 Comparing Executions

Model comparison, in general, is an important task in order to obtain the benefits of MDE. Being able to track the evolution of a model through its life-cycle helps understanding a system's evolution. The EMF Compare project [5] has been initiated in 2006 with the goal to provide a simple support for model comparison and model merging. It is designed in a modular way, making all components replaceable and customizable. The EMF Compare model comparison process is separated into two phases; the *matching* phase, performed by a match engine, and the *diffing* phase, performed by a diff engine. During the match phase, elements from one model are

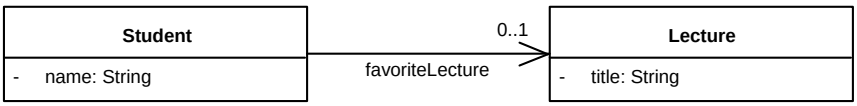

Figure 5.6: Sample fUML class diagram illustrating the comparison of executions

matched with elements from the other model. By matching one element to another, the system is being told that specific elements in two models correspond to each other and that, during the *diffing* phase, discovered differences between those are significant and worthy to report. The default match engine, provided by EMF Compare, tries to match an element by traversing the other model until a match was found. Two elements are matched through their identifier (if they have one) and through a distance algorithm for all elements without identifier [2].
While this comprehensive, generic matching method is suitable for most use cases, it is not optimal for the comparison of the captured execution event tracing models. Therefore, a custom match engine, which replaces the default EMF Compare match engine, was developed. Its functionality is described in the following.
After its completion, the result of the *matching* phase is refined in the *diffing* phase. By iterating over all matching elements, the differences among them are computed. If only one side of a match element has an object assigned (i.e. it is unmatched), it is determined that an element has been added or deleted from one model in comparison to the other.

Custom Match Engine

There are different strategies to verify whether two programs are executed equally. A naive approach would check whether the properties of all created objects have the same values after the end of the execution. This, however, leaves too much space for uncertainty and the chances to miss significant differences between the two executions are high. Another strategy would be to compare every single event of the execution and raise an error if the two events are not equal. Since there are, however, scenarios where the execution order of actions is irrelevant, as for instance in models with multiple candidates for initial nodes, the method would result in a lot of false positive errors.

The strategy of the custom match engine is based on the latter method but instead of directly comparing all events, the `FeatureValueEvents` of each execution are assigned to their corresponding `ExtensionalValue` in a first step. This results in groups of `ExtensionalValueEvents`, i.e, `ExtensionalValues` with a list of events that represent all modifications performed on them in their chronological order, which is the basis for the further matching process.
In the first step, the match engine matches the `ObjectStoreItems` and their child elements. Two `ObjectStoreItems` do match if their type property references the same UML class and both contain the same `Features` with all of them referencing the same UML property. The order of the `FeatureValue` items within the `ExtensionalValue` is not relevant, nor is the

[2] `https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html#Match_2`

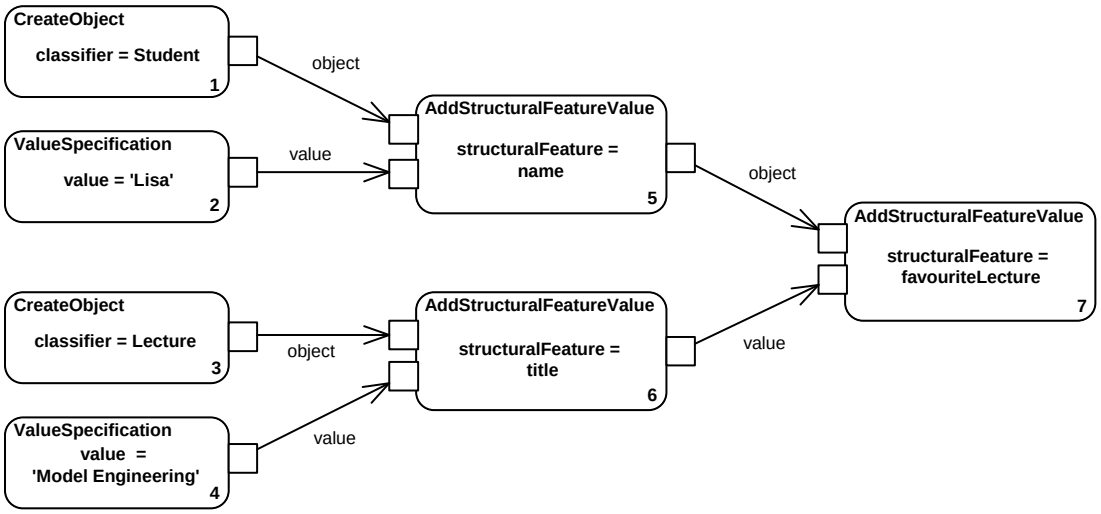

Figure 5.7: Sample fUML activity diagram illustrating the comparison of executions

order of `ObjectStoreItems` within the `ObjectStore`. Under the assumption that for every item from the `ObjectStore` of the Reference Storage (i.e., of the execution of the fUML model), a matching item in the Comparator Storage (i.e., of the execution of the generated code) can be found, this step verifies that the code generator correctly generated all properties defined by the class diagram in the static part of the fUML model.

In a second step, the match engine matches all `FeatureValueEvents` of each `ExtensionalValue`. Since the order of `FeatureValueEvents` is substantial, it is tried to establish a match between the first event of an `ExtensionalValue` of the Reference Storage and the first event of the matching `ExtensionalValue` captured in the Comparator Storage, the second event to the second event and so on. Two `FeatureValueEvents` are equivalent if they reference the same `Feature` and have the same `ExtensionalValueEventType`. If two `FeatureValueEvents` are equal, a match is established between them. If, however, the two `FeatureValueEvents` are not equal, only one side of the match element can be set, which will be reported as an added or removed item in the diffing phase.
In the next step, all `FeatureValues` of the matched `FeatureValueEvents` are matched; two `FeaturesValues` match if they reference the same UML property and their `values` and their `position` elements match, i.e. if the have the same values and position. A `FeatureValueEvents` is considered to be equal if a match for every FeatureValue and their Values is found.
By checking the equality of the corresponding events, it is assured that in both models, all `ExtensionalValues` are equally modified from their creation until the end of execution.

Example 1

Figure 5.7 shows a simple activity diagram; an object of the type `Student` is created (1), the value Lisa is specified (2) and assigned to the property `Name` (5). A second object of the type

`Lecture` is created (3), the value ModelEngineering specified (4) and assigned to the property `Title` (6). Finally, the property `FavouriteLecture` of the student object is set to the lecture object (7).

There are various different orders in which the fUML virtual machine or the code generator can process the actions. In regard to the condition, that all incoming pins of an action need input values before the action can be executed, the only restrictions are that (1) and (2) are processed before (5), (3) and (4) are processed before (6), and (5) and (6) are processed before (7).

Let the execution order of the fUML machine be

$$O1 = \{1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7\}$$

and the processing order of the code generator be

$$O2 = \{4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 7\}$$

The generated code for the order O2 is shown in Listing 5.2

```
4: // No code generated
3: Lecture l1 = new Lecture();
2: // No code generated
1: Student s1 = new Student();
6: l1.setTitle("ModelEngineering");
5: s1.setName("Lisa");
7: s1.setFavouriteLecture(l1);
```

Listing 5.2: Code generation result for the activity shown in Figure 5.7 executed in order O2

The left part of Table 5.1 contains the events caused by the execution of the model with the fUML virtual machine in the order O1. The right side of the table contains the events caused by the execution of the generated code in Listing 5.2.

Step	O1	Events	O2	Events
1	1	EVE s1:Student	4	-
2	2	-	3	EVE l1:Lecture
3	5	FVE s1.Name = Lisa	2	-
4	3	EVE l1:Lecture	1	EVE s1:Student
5	4	-	6	FVE l1.Title = 'Model Eng.'
6	6	FVE l1.Title = 'Model Eng.'	5	FVE s1.Name = Lisa
7	7	FVE s1.FavouriteLec. = l1	7	FVE s1.FavouriteLec. = l1

Table 5.1: Extensional value events (EVE) and feature value events (FVE) caused by execution of the activity shown in Figure 5.7 in order O1 and O2

In the first iteration, the custom match engine tries to match all `ExtensionalValues`. Every created object on the left side (Student and Lecture) exists on the right side and no object on the right side remains. For this example, it's assumed that the `FeatureValue` elements of

Step	Events
1	EVE s1:Student
3	FVE s1.Name = Lisa
7	FVE s1.FavouriteLec. = l1
4	EVE l1:Lecture
6	FVE l1.Title = 'Model Eng.'

Step	Events
4	EVE s1:Student
5	FVE s1.Name = Lisa
6	FVE s1.FavouriteLec. = l1
3	EVE l1:Lecture
7	FVE l1.Title = 'Model Eng.'

Table 5.2: Extensional value events (EVE) and feature value events (FVE) caused by execution of the activity shown in Figure 5.6 in order O1 and O2 grouped by their extensional values.

both objects are equal by which it is verified that the structural part of the model was generated correctly.

In the next step, all events with the same `ExtensionalValues` are grouped together (i.e. all `FeatureValueEvents` are grouped by their corresponding `ExtensionalValueEvents`). The result of grouping the events is showin in Table 5.2. The events are then matched based on their index. When all matched pairs are equal, which is the case in this example, the generated code behaved equivalently as the fUML model although the actions were not processed in the same order.

Example 2

In this example, the code generator processes the actions of Figure 5.7 in the order

$$O3 = \{4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow 5\}$$

which is an invalid order, since (7) is processed before (5). The generated code for the order O3 is shown in Listing 5.3.

```
4: // No code generated
3: Lecture l1 = new Lecture();
2: // No code generated
1: Student s1 = new Student();
6: l1.setTitle("ModelEngineering");
7: s1.setFavouriteLecture(l1);
5: s1.setName("Lisa");
```

Listing 5.3: Code generation result for the activity shown in Figure 5.7 executed in order O3

Table 5.3 contains the events caused by execution of the activity shown in Figure 5.7 in order O1 and O3.

After assigning the `FeatureValueEvents` to their corresponding `ExtensionalValue` (see Table 5.3), the pairwise comparison shows that the events on position 2 and 3 are not equivalent. The two executions are therefore unequal and the result of the code generation is considered invalid.

Step	O1	Events	O3	Events
1	1	EVE s1:Student	4	-
2	2	-	3	EVE l1:Lecture
3	5	FVE s1.Name = Lisa	2	-
4	3	EVE l1:Lecture	1	EVE s1:Student
5	4	-	6	FVE l1.Title = 'Model Eng.'
6	6	FVE l1.Title = 'Model Eng.'	7	FVE s1.FavouriteLec. = l1
7	7	FVE s1.FavouriteLec. = l1	5	FVE s1.Name = Lisa

Table 5.3: ExtensionalValueEvents (EVE) and FeatureValueEvents (FVE) caused by execution of the activity shown in Figure 5.7 in order O1 and O3

Step	Events
1	EVE s1:Student
3	FVE s1.Name = Lisa
7	FVE s1.FavouriteLec. = l1
4	EVE l1:Lecture
6	FVE l1.Title = 'Model Eng.'

Step	Events
4	EVE s1:Student
6	FVE s1.FavouriteLec. = l1
5	FVE s1.Name = Lisa
3	EVE l1:Lecture
7	FVE l1.Title = 'Model Eng.'

Table 5.4: ExtensionalValueEvents (EVE) and FeatureValueEvents (FVE) caused by execution of the activity shown in Figure 5.6 in order O1 and O3 grouped by their extensional values

Alternative Equivalence Criteria

The presented code verification approach is based on the comparison of feature modifications of matched objects captured in instances of the EET metamodel. However, various alternative equivalence criteria exist that can be used to verify the correctness of the generated code in a different way. In EMF Compare, different compare strategies can be easily realized by implementing custom match engines like the one presented in this section.

A possible alternative equivalence criteria could be to require that all actions of a UML activity have to be executed in the same order in the generated Java code. The event model of the fUML virtual machine [31] provides such information by emitting an *ActivityNodeEntryEvent* when the execution of a node is started and an *ActivityNodeExitEvent*, when the execution of the node is completed. Many fUML actions, e.g., the *ValueSpecificationAction* or the *ReadSelfAction*, provide values for successive nodes but do not directly result in any Java code being created when processed by the code generator. The generated code would therefore have to be extended by a kind of meta data that indicates the beginning and the end of an action, which enables the monitoring component to capture and provide corresponding events. Additionally, the developed EET metamodel would have to be extended accordingly in order to be able to store such events. This approach would allow very detailed investigations of the runtime behavior and could be utilized if the evaluation of the correctness of the chronological order of node

executions is of high importance.
A different equivalence criteria would be to require that the Java code generated for any UML activity produces the same output as the execution of the activity by the fUML virtual machine. The class *Input* and *Output* and *ParamterInput* and *ParameterOutput* of the trace model of the fUML virtual machine [31] provide such information. In order to gain similar information of execution of the generated code, the monitoring component could be easily extended to capture the input and return values of methods, which could then be matched to the inputs and outputs of activities. However, to capture the inputs and output of activity nodes, the generated code would have to be enriched by such meta data since the input out output values of activity nodes are often not directly reflected in the generated code. The EET metamodel would also have to be extended to provide the means to store such additional information. This equivalence criteria could be valuable in situations where the analysis and evaluation of the generated object flow is of high importance.

CHAPTER 6

Evaluation

The aim of this thesis was do develop a code generation approach that generates Java code from an input fUML model, which, when executed, behaves equally as the execution of the input model carried out by the fUML virtual machine. The developed code generation approach was presented in Chapter 4. In Chapter 5, a process to verify the correctness of code generated with the developed fUML code generator is presented. This chapter is concerned with the evaluation of the developed code generator. In the evaluation, the verification process introduced in Chapter 5 has been applied. In this chapter, we first introduce the evaluation method and then describe the conducted evaluation in detail. The developed test models, the case study and the evaluation results are available online. Further information about the public repository are provided in Appendix A.

6.1 Overview

The goal of the evaluation carried out as part of this thesis was to answer the following two research questions:

- Does the developed code generator transform an fUML input model into Java code that behaves equally when executed as the execution of the input model carried out by the fUML virtual machine?
- If inequalities are found, what are the reasons for differences between the executions and which modifications of the generator would have to be made for the executions to behave equally?

To answer these questions, a comprehensive test suite with fUML models testing different scenarios has been built and a case study on a larger fUML model was conducted. In the test suite, a wide set of sample input models was used to test the correctness of individual fUML actions supported by the fUML code generation. The sample input models consist of minimal

class and activity diagrams, in which different test scenarios are used to set up isolated testing conditions for a certain fUML action. All test models are compiled into a test suite. The detailed set up of the test suite and the results of the tests are described in Chapter 6.2. In the case study, a comprehensive fUML model modeling an online store, consisting of 13 classes and 25 activities is used to validate the correctness of the code generator against a more real-world sized model. A detailed introduction to the used input model as well as the results of the evaluation of its generated code are presented in Chapter 6.3.

6.2 Test Suite

The test suite consists of a sets of tests for every fUML action supported by the code generator. Every test case tests the correctness of a specific fUML action under certain settings and given input values. The following list contains a brief description of all tests conducted. If the test failed, i.e. the execution of the generated code and the execution performed by the fUML virtual machine were not equal, an explanation for the differences is given.

Object Actions

CreateObject

Test case	*Description*	*Test result*
Test 1	Creates an object of a certain type	passed

DestroyObject

Test case	*Description*	*Test result*
Test 1	Creates an object of a certain type and immediately destroys it afterwards	**failed**
The virtual machine emits an extension value event of the type DESTRUCTION when an objects is destroyed by the DestroyObject action. Since Java does not support object deconstruction, no corresponding event can be captured from the execution of the generated code.		
Test 2	Creates two objects, adds the second object as value to a multivalued, complex typed feature of the first object and destroys the added object	**failed**
The generated code correctly removes all items from the multivalued feature, however, since no events are captured for object destructions as described in Test 1, the event models differ.		

ValueSpecification

Test case	*Description*	*Test result*
Test 1	Specifies a string literal and sets it as value of an feature of a created object	passed

ReadSelf

Test case	*Description*	*Test result*
Test 1	Creates an object, calls an operation on itself and modifies a feature value it owns	passed

TestIdentity

Test case	*Description*	*Test result*
Test 1	Specifies two primitive values, tests whether the to values are identical and assigns the result to a Boolean property	passed
Test 2	Creates two objects, test whether they are identical and assigns the result to a Boolean property	passed

ReclassifyObject

Test case	*Description*	*Test result*
Test 1	Creates an object of a subtype and casts it to its super type	**failed**

In fUML, every object is characterized by a set of types. When an object gets reclassified, two extensional value events are emitted; one notifying about the removed types and one about the added types. The code generator explicitly casts the provide object and assigns it to new variable of the new type. This triggers the ObjectInitialized point cut, resulting in a single extensional value event.

ReadIsClassifiedObject

Test case	*Description*	*Test result*
Test 1	Creates one object of a type `X` and a second object of a type `Y`, which is a subtype of `X` and checks whether the second object is of type `X`	passed
Test 2	Creates one object of a type `X` and a second object of a type `Y`, which is a subtype of `X` and checks whether the second object is directly of type `X`	passed

ReadExtent

Test case	*Description*	*Test result*
Test 1	Creates two object of a type `X`, retrieves them using a ReadExtentAction and modifies a certain feature value of all of them	passed

Reduce

Test case	*Description*	*Test result*
Test 1	Creates an object, adds two integer values to a multivalued property, calls a reduce operation on these values and assigns the result to an integer property	passed

Structural Feature Actions

AddStructuralFeatureValue

Test case	*Description*	*Test result*
Test 1	Creates an object and sets the value of a primitive typed feature of the object	passed
Test 2	Creates two objects and sets the value of a complex typed feature of the first object to a reference to the second object	passed
Test 3	Creates two objects and adds the second object as value to a multivalued, complex typed feature of the first object	passed
Test 4	Creates an object and adds two primitive values at a certain position to a multivalued feature of the object	passed
Test 5	Creates an object and adds two objects at a certain position to a multivalued, complex typed feature of the object	passed

ReadStructuralFeature

Test case	*Description*	*Test result*
Test 1	Creates an object and sets the value of a primitive typed feature of the object. Creates a second object, reads the value of the feature of the first object and assigns it to one of its own features	passed
Test 2	Creates an object and adds multiple complex values to a multivalued feature of the object. Creates a second object, reads the values of the feature of the first object and assigns it to one of its own features	passed

RemoveStructuralFeature

Test case	*Description*	*Test result*
Test 1	Creates an object, sets the value of a primitive typed feature of the object and removes the value of the feature	**failed**
The generated code removes the value by setting it to `null`. Since the storage factory can not distinguish between the removal of a value and an intentional null-value assignment, a feature value event of the type VALUE_ADDED is added to the comparator storage. The fUML virtual machine, however, emits a VALUE_REMOVED event when it removes a feature value, which results in the executions to be different.		
Test 2	Creates two objects, sets the value of a complex typed, single-values feature of the first object to a reference to the second object and removes the second object as value from the feature	**failed**
See Test 1		
Test 3	Creates two objects, adds the second object as value to a multi-valued, complex typed feature of the first object and removes the second object from the feature	passed
Test 4	Creates two objects, adds the second object as value to a ordered, multivalued, complex typed feature of the first object and removes the second object from the feature at a specified position	passed

ClearStructuralFeature

Test case	*Description*	*Test result*
Test 1	Creates an object, sets the value of a primitive typed, single-valued feature of the object and clears the same feature	**failed**
See Test 1 of RemoveStructuralFeature		
Test 2	Creates two objects, sets the value of a complex typed, single-valued feature of the first object to a reference to the second object and clears the feature	**failed**
See Test 1 of RemoveStructuralFeature		
Test 3	Creates two objects, adds the second object as value to a multi-valued, complex typed feature of the first object and clears the feature	passed

Link Actions

CreateLink

Test case	*Description*	*Test result*
Test 1	Creates two objects and creates a link between the objects. Since both link ends are navigable, a reference to the respective other object is assigned to a feature of both objects	passed
Test 2	Creates two objects and creates a link between the objects. Both link ends are navigable, one of the link ends is multivalued.	passed
Test 3	Creates two objects and creates a link between the objects. Both link ends X and Y are navigable and link end Y is multivalued. An *insertAt* value is provided for link end Y. The object is therefore inserted at a specific index of the referenced collection. Additionally, the *isReplaceAll* property of link end Y is set to true, resulting in all objects of the collection to be destroyed before the new object is added.	passed

ReadLink

Test case	*Description*	*Test result*
Test 1	Creates two objects, creates a link between the two objects, reads the created link and assigns the retrieved value to a new specific property	passed

DestroyLink

Test case	*Description*	*Test result*
Test 1	Creates two objects, creates a link between the objects and destroys the link. Both link ends are navigable, therefore both feature values to which the references were assigned are removed	passed
Test 2	Creates two objects, creates a link between the objects and destroys the link. Both link ends are navigable but one of the link ends is multivalued. The reference of the multivalued link end is therefore added to a collection.	passed
Test 3	Creates two objects, creates a link between the objects and destroys the link. Link end X and Y are navigable and link end Y is multivalued. An *insertAt* value is provided for link end Y. The object is therefore first inserted and then destroyed at a specific index of the referenced collection.	passed

Communication Actions

CallBehavior

Test case	*Description*	*Test result*
Test 1	Creates an objects, creates two Integer values and calls a behavior, which performs an operation on the Integer values. The result of the behavior is assigned to a specific property.	passed

CallOperation

Test case	*Description*	*Test result*
Test 1	Creates an object, creates a string value and calls an operation on the object. The result of the operation is assigned to a property of the object.	passed

Control Nodes

ForkNode

Test case	*Description*	*Test result*
Test 1	Creates an objects and passes the object to a fork node, which duplicates the object for all outgoing edges. One property of the object is modified by each action connected to the outgoing edges of the fork node.	passed

JoinNode

Test case	*Description*	*Test result*
Test1	Creates an object and an Integer value. The JoinNode waits for both actions to be performed and then assigns the Integer value to a property of the object.	passed

DecisionNode

Test case	*Description*	*Test result*
Test 1	Creates a boolean value, passes the boolean value to a decision node and creates an object depending on the boolean value.	passed
Test 2	Creates an integer value and passes the value as input to an opaque behavior defined as decision input of the decision node. Creates an object depending on the result of the decision node.	passed

MergeNode

Test case	*Description*	*Test result*
Test 1	Creates an integer value, forks the value to two decision nodes and merges the outgoing control flows into a single flow. Modifies the incoming value depending on the execution of the decision node.	passed
Test 2	Creates an integer value, forks the value to two decision nodes and merges three outgoing control flows into a single flow. Modifies the incoming value depending on the execution of the decision node.	passed
Test 3	Creates an integer value, forks the value to two decision nodes with overlapping conditions and an else condition. Merges four outgoing control flows into a single flow. Modifies the incoming value depending on the execution of the decision node.	passed

Structured Activity Nodes

ExpansionRegion

Test case	*Description*	*Test result*
Test 1	Creates two objects and assigns two string values to a structural features of each object. Both objects are passed to an expansion region. The region processes both objects and assigns the string values to a different structural feature of the objects.	passed

Summary

The results of the test suite show that the majority of fUML actions could be mapped to equally behaving Java code. Out of 41 tested scenarios, seven failed. All of the seven failed tests were a result of incorrect event reporting by the monitoring component of the Java code. In particular, for the test case testing the *ReclassifyObject* action, only a single event with the newly assigned type is sent to the storage factory while the fUML virtual machine reports both the old and new type of the reclassified object. The actions *DestroyObject*, *RemoveStructuralFeature* and *ClearStructuralFeature* have in common that they explicitly remove an object or a value from an object, which is reported as such to the storage factory. Java does not provide a possibility to deconstruct objects and values. The code generator translates these actions into a null-value assignment (or the assignment of a default value, if the affected property is of a primitive type), which is reported as value assignments to the storage factory. Given the implemented equivalence criteria, this results in the executions being reported unequal. However, a more detailed

examination of the failed tests has shown that despite the utilized equivalence criteria not being met, no differences in the behahivor of the executions were observed.

6.3 Online Store Case Study

After the isolated evaluation of the supported fUML actions, a case study was conducted that tested the code generator's abilities in an more realistic environment. While the test suite provides valuable information about the translation of single fUML actions and enabled the code generator to be developed in a test-driven way, it is not suited to evaluate the correctness of the code generator in a wider scope. In the following, a description of the fUML model and the test scenarios, which were used to conduct the case study, is given.

Model Description

The investigated PetStore.uml model describes the structure and behavior of an online store system in which a users can perform various tasks like login, find products and create orders. The model, developed by the Business Informatics Group at the TU Wien, was originally created to serve as case study for the extension of the fUML virtual machine [31].
The structural model, depicted in Figure 6.1, consists of 13 classes. The entities processed by the online store are described by the classes below the dashed line while classes that offer services to control the behavior of the system are shown above the line. A session (class Session) is identified by a unique id and is associated with a customer (class Customer). A customer has a shopping cart (class Cart) in which the customer can add cart items (class CartItem) by choosing items from the offered products (classes Item and Product). When a user checks out, an order (class Order) is created, that holds an order line (class OrderLine) for every item purchased.
The behavior of the system is described by the activities defined in the classes CustomerService, CatalogService, OrderService, EntityManager and ApplicationController. The ApplicationController provides all services that manage user interactions, the CatalogService provides methods with which items and products from the product catalog can be found. To authenticate a customer, services from the CustomerService are used. To find and persist entities, services by the EntityManager are used. To give an impression about the size and complexity of the activities, the activity getCartItem of the class OrderService and the generated code for this activity are shown in Figure 6.2 and Listing 6.1 respectively.
The Application Host (class AppHost, not depicted in Figure 6.1) contains a set of test scenarios. The scenarios simulate typical user stories of the online store like logging into the store, adding items to the cart and ordering items. These scenarios are use to conduct the case study. Each scenario is executed by the fUML virtual machine and compared with the execution of the generated code. The following list contains a brief description of the scenarios executed with the results of the compared executions. The tables below the descriptions summarize the numbers of objects created and modified during each execution.

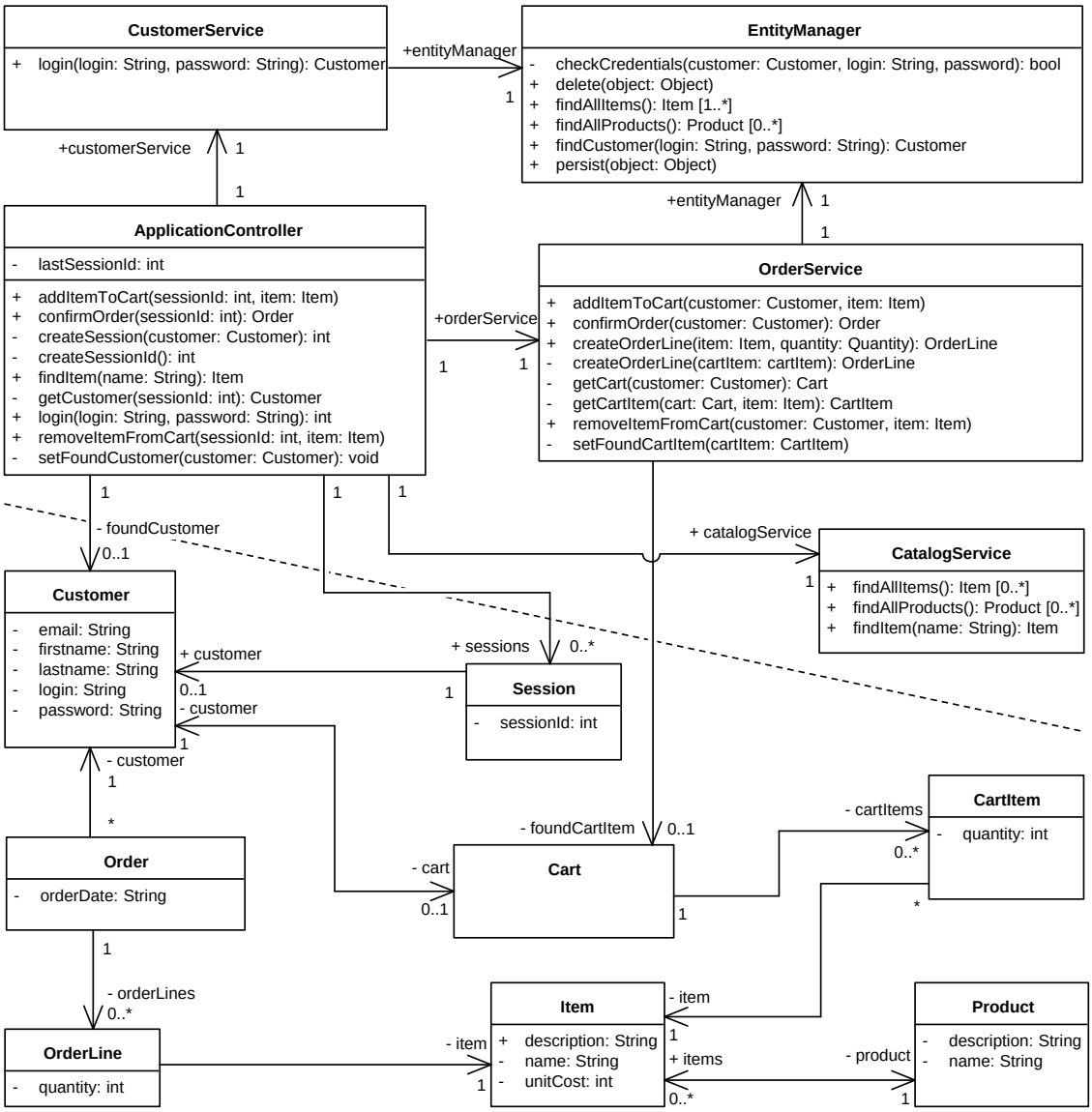

Figure 6.1: Class diagram of the online store case study

In order to get an impression of the size and complexity of the activity diagrams considered by the case study, the number of activity nodes, activity edges and actions that were processed by the code generator are listed below.

- Activities *49*
- Nodes *1.228*
- Average number of nodes per activity *25*
- Edges *364*
  - Object flows *326*
  - Control flows *38*
- Control nodes *50*
- Object actions *87*
- StructuralFeature actions *54*
- Link actions ..... -
- Communication actions *80*

Test Scenarios

Scenario 1 - Successful login

A user tries to login to the online store. The provided password and username are correct and the user can be successfully logged in. A new sessionId is returned by the login method of the ApplicationController and assigned to a property.

	fUML virtual machine	*Generated code*
Activities executed	16	-
Nodes executed	97	-
Objects created	8	8
Object modifications	18	17*

Differences

* *The virtual machine emits an additional event when removing the value of a primitive feature. As described in Section 6.2, no such event can be captured from the generated code. Apart from this, the executions are equal.*

Scenario 2 - Unsuccessful login

A user tries to login to the online store. Since a wrong password is provided, the user can not be logged in to the system. The login method of the ApplicationController returns -1.

	fUML virtual machine	*Generated code*
Activities executed	14	-
Nodes executed	74	-
Objects created	7	7
Object modifications	11	12*

Differences

* *In the generated code, the result of the method, that looks up the requested customer object, is assigned to a structural feature, regardless of its value (in this scenario* `null`*). During the execution carried out by the virtual machine, the* AddStructuralFeatureValue *action that assigns the value of the corresponding activity is not executed since the required token is missing. Both executions, however, correctly return the Integer value* `-1`*.*

Scenario 3 - Search for existing item in catalog

Passes a search expression to the catalog service and returns the first item with a name equal to the search expression. The catalog service in this scenario finds and returns an item.

	fUML virtual machine	*Generated code*
Activities executed	12	-
Nodes executed	53	-
Objects created	8	8
Object modifications	10	10

Differences
none

Scenario 4 - Search for non-existing item in catalog

Passes a search expression to the catalog service and returns the first item with a name equal to the search expression. The catalog service does not find a matching item in this scenario and therefore, does not return a value.

	fUML virtual machine	*Generated code*
Activities executed	12	-
Nodes executed	53	-
Objects created	8	8
Object modifications	10	10

Differences
none

Scenario 5 - Add items to cart

The user logs in to the system and adds three items to the shopping cart. When an item is added, that already exists in the cart, which is the case in this scenario, the quantity property of the respective item object is increased.

	fUML virtual machine	*Generated code*
Activities executed	72	-
Nodes executed	396	-
Objects created	14	14
Object modifications	50	48*

Differences

* *Besides minor differences in how values are removed from features, the executions are equal. One notable difference was found in the creation of a link between two object. While the fUML standard recommends the use of the* CreateLink *action, the fUML virtual machine as well as the code generator also support the* AddStructuralFeatureValue *action to create a link between two features. The virtual machine sets the reference from the link source to the target and vice versa, which results in the reporting of two* FeatureValueEvents*. The code generator, however, only sets the references to the navigable link ends, which, in the case of this scenario, results in the reporting of a single* FeatureValueEvent*.*

6.4 Summary

In this chapter, a test suite and a case study to evaluate the correctness of the developed code generation approach have been presented. The elaborated code verification framework introduced in Chapter 5 automatically invokes the code generation, the execution of the fUML model, the execution of the generated Java code and the comparison of the execution event tracing models, which allowed an efficient implementation of the test suite and the case study. This indicates

that the code generator's *Testability* requirement has been fully archived. The results of the test suite show that the majority of the supported fUML actions can be directly translated into equally behaving Java code. The deviations between the events captured during the execution carried out by the fUML virtual machine and the events captured during the execution of the generated code were caused by differences in the way object modifications are reported to the storage factory and were found to not impact the bahvior the executions. The results of the test suite also show that the code generator's *Completeness* requirement to provide complete support of all UML activity concepts has almost been achieved. fUML actions for which over the course of this thesis no Java mapping was elaborated or for which only limited support is provided are discussed in Chapter 4.6. The code generator was developed using state of the art model-to-text technologies. This allows the code generator to be easily extended in order to provide support for the missing fUML actions in the future. It also provides means to elaborate mappings for fUML actions for target languages other Java. Thereby, the code generator's requirement for *Flexible* wrt. the target language has been met.
The conducted case studies show that the code generator for fUML models works in a reliable manner. All detected differences between the executions of the generated code and the executions carried out the fUML virtual machine can be traced back to different approaches in reporting the removing of feature values and different assumptions for undefined executions semantics (e.g. create links with AddStructuralFeatureValueAction). It further confirmed that the developed code validation framework works as intended. However, the interpretation of the results of complex activities with a very high number of actions turned out to be rather difficult. fUML models containing actions that cannot be directly mapped to a corresponding Java code fragment (identified by the failed tests in the test suite) ultimately result in the comparison component to report the executions as being non-equal. The identification of the cause of the inequality often required extensive manual investigation. Possible improvements to this circumstance are discussed in Section 7.2.

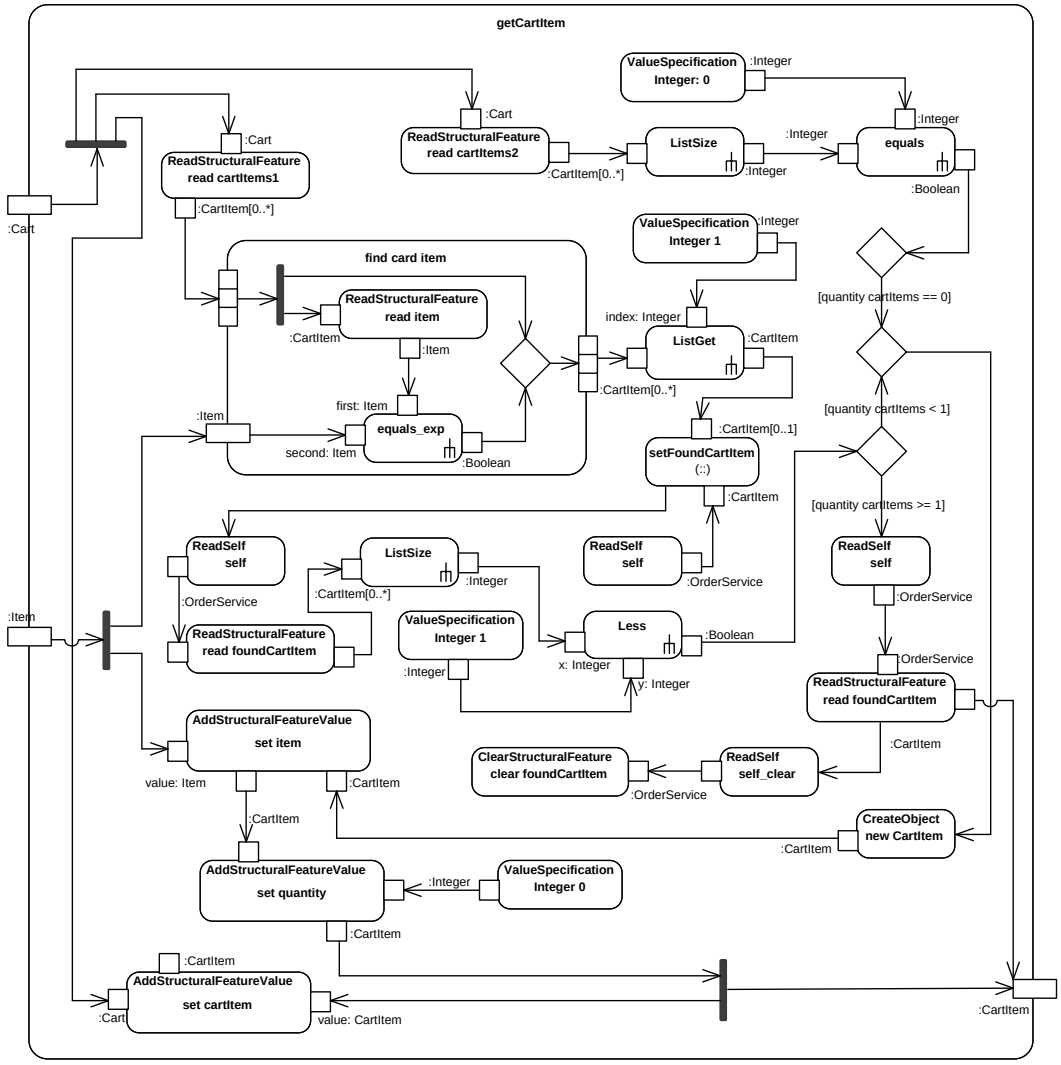

Figure 6.2: Activity diagram getCartItem of the online store case study

```
private CartItem getCartItem(Cart cart, Item item) {

   ...

   while (it_var1.hasNext()) {
      CartItem find_cart_item_var1 = it_var1.next();
      Item var5 = find_cart_item_var1.getItem();
      Boolean var6 = equals(var5, item);
      if (var6.equals(new Boolean(true))) {
         var4.add(find_cart_item_var1);
      }
   }

   Boolean var7 = equals(var3, new Integer(0));
   CartItem var8 = var4.size() > new Integer(0)
      ? ((java.util.ArrayList<CartItem>)var4).get(new Integer(0))
      : null;
   this._setFoundCartItem(var8);
   CartItem var9 = this.getFoundCartItem();
   Integer var10 = var9 instanceof java.util.Collection<?>
      ? ((java.util.Collection<?>) var9).size()
      : (var9 == null ? 0 : 1);
   Boolean var11 = var10 < new Integer(1);

   if (var11.equals(new Boolean(false))) {
      CartItem var12 = this.getFoundCartItem();
      this.setFoundCartItem(null);
      cartItem = var12;
   }

   if (var11.equals(new Boolean(true)) || var7.equals(new
       Boolean(true))) {
      CartItem var13 = new CartItem();
      var13.setItem(item);
      var13.setQuantity(new Integer(0));
      cartItem = var13;
      cart.getCartItems().add(var13);
   }

   return cartItem;
}
```

Listing 6.1: Code generation result for the activity getCartItem of the online store case study depicted in Figure 6.1

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented a code generation approach that translates fUML models into executable Java code. To gain detailed information about the execution of the generated code, a component was introduced that is concerned with monitoring the execution of the generated code. To evaluate the correctness of the generated code, a code verification component was presented, that compares the information provided by the monitoring component and the execution monitoring of the fUML virtual machine. The presented components aim to contribute to an essential feature of MDE, the automatic code generation, for fUML. The contributions compiled in the course of this thesis are summarized in the following.

Code Generator for fUML

Code generation in general is an important part for MDE as it enables the necessary transformation from a high level of abstraction to a lower level. The transformation from UML models, that describe executable systems, into an executable target language, has been particularly tedious for many years due to the lack of well defined execution semantics for behavioral UML modeling concepts. An important requirement for the code generator is to not rely on individual interpretations of a language specifications.
With the publication of the fUML standard, which defines execution semantics for a subset of UML, the basis for accurate transformations, like the code generator presented in this thesis, was created. The semantics of the language are defined operationally in terms of a virtual machine, which enables the execution of models compliant to fUML. However, the presented code generator has no dependencies to an implementation of the fUML virtual machine, but instead can operate independently, i.e. the target code can the produced without the fUML virtual machine to interpret the fUML model during the transformation.
The code generator presented in this thesis produces Java code, however, the components were designed in an easily extensible form so that the currently supported target language can be ex-

changed quite easily. The generator supports all fUML language concepts except a few actions listed in Chapter 4.6. By that, support for one of the key features of MDE, the automated transformation from a higher level of abstraction to a lower level, is enabled for fUML. Thereby, an increase in productivity and efficiency for users of fUML can be achieved.

Code Monitoring

To evaluate the correctness of the generated code, a monitoring component was introduced that records the creation and modification of objects during the execution of the generated code. The monitoring component was designed in such a form that it provides the necessary information even after a modification or extension of the current code generator component. The implementation of code monitoring component is tightly coupled to the utilised target language, in the case of this thesis Java. However, the developed components can be easily adapted for a different target language.
The behavior of the fUML virtual machine was used as a reference for checking the correctness of all generated code artefacts. To access runtime information about the processing of fUML models, an extension of the fUML virtual machine [31], in form of an event model, was used. The event model is based on an abstract representation of the execution of an fUML model. Changes in the runtime state trigger corresponding events, which provided the basis for the evaluation of the generated code. However, certain language concepts of fUML (e.g. creation of link objects) may not be available in the utilized target language. It was therefore necessary to introduce a target platform specific component that acts as adapter between the runtime information captured by the code monitoring component and the event model provided by the fUML virtual machine. The component developed over the course of this thesis addresses this necessity by transforming the events provided by the fUML event model in such a way that they are directly comparable to the events captured during the execution of the generated Java code.

Comparison Engine

The developed comparison component processes the runtime information gathered from the execution of the generated code and the execution of the corresponding fUML model, represented as instances of an execution event tracing metamodel. The goal of the utilized comparison strategy is to compare two executions as detailed as possible, while avoiding too many false positive errors. It has been shown that fUML models that define no control flow may be processed in a very different execution order while behaving equally. Therefore, a one-by-one comparison between two executions has proven to be not effective. Instead, the modifications of matching features of every object created during runtime are compared pair-wise. If every feature modification of the first execution event tracing model can be matched to a feature modification of the second model, two executions are considered to be equal.
The comparison engine is implemented as custom EMF Compare match engine. A great advantage of this solution is that the component can be easily integrated into the EMF ecosystem. Also, the match engine can be easily adapted to apply different equivalence criteria to determine whether two executions are equal.

Evaluation Results

The results of the tests conducted during the evaluation of the developed code generator show that for most of the supported fUML actions, equally behaving Java code could be generated. A test suite, which contains a variety of test scenarios in which the supported fUML actions were tested with different inputs and under different conditions, served as basis for the evaluation process. The test scenarios revealed a few fUML actions, for which the generated code behaved differently than the execution carried out by the fUML virtual machine. Investigations showed that the failed tests were caused by differences in the way object modifications are reported to the storage factory. The conducted case study has shown that the code generator also works reliably for comprehensive, more real-world sized fUML models. Detected differences in the result of the case study could be traced back to different approaches in reporting the removing of feature values and different assumptions for undefined executions semantics. The differences, however, were found to not impact the bahvior of the execution. It was also shown that the developed code validation framework works as intended.

7.2 Future Work

In the following, possible improvements to the presented components are discussed. Most of the suggestions are concerned with the capabilities of the presented components, such as support for additional target languages. The suggestions arose partly during the development of the components, and partly from the results of the case studies carried out.

Provide better support for complex transformations

The basic architecture of the code generator is simple yet very effective. When an input model is processed, a set of potential starting nodes is identified and processed recursively. Depending on the type of the processed node, a corresponding generation method is invoked that adds the generated code to the result. In its current state, the generation method for each action is completely self-sufficient and unaware of its context. It has only access to the currently processed node but has no information about the nodes that caused its invocation. These limitations have been found to be a hindrance in many cases. Ad hoc solutions helped to overcome these limitations, however, in order to provide support for more advanced transformations that depend on information beyond the currently processed node, it might be necessary to fundamentally revise the basic operation principles of the code generator.

Separation of code generator and target language

The code generator for fUML developed in this thesis is written with the M2T language Xtend[1]. It fully supports the template engine Xpand[2] that allows a clean separation between templates

[1]The Xtend documentation is available online at `http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.xtend.doc%2Fcontents%2FXtend.html`, accessed 10-12-2016

[2]The reference for the Xpand syntax is available online at `http://git.eclipse.org/c/m2t/org.eclipse.xpand.git/plain/doc/org.eclipse.xpand.doc/manual/xpand_reference.pdf`,

of the target language code and code from the code generator and provides good readability. As described above, every supported fUML action has a corresponding generation method within the generator. In its current version, the templates of the code of the target language are directly located within the generation methods.
It would be desirable to provide an even cleaner separation of the generation methods and the target language code, and thereby simplify the extension of the code generator for additional target languages. One possibility would be to insert an abstraction layer between the generator and the target language code templates and outsource the actual code blocks of the target language to, for example, Java ResourceBundles[3]. The code generator could then load the selected target language at runtime and access the required code templates.

Separation of monitoring component and event storage

In its current state, the monitoring component is tightly coupled to the storage factory, which processes the captured events from the code execution and converts them into an instance of the execution event tracing metamodel. To provide the functionality of the storage factory to monitoring components that capture execution events of different target languages, it would be necessary to convert the storage factory into a service that exposes methods that enable the user to add entries to the execution event tracing. The service could then be used by other aspect-orientated programming languages like AspectC[4] for the language C++ or PostSharp[5] for languages like C# and Visual Basic.

Eliminate differences between generated code and fUML model

As described in Chapter 4.5, it was not always possible to directly map an fUML action to a corresponding Java code fragment. For instance, fUML allows the deconstruction of objects, a concept that is not provided by Java. Therefore, a model containing an action that destroys an object will report a corresponding destroy event when executed by the fUML virtual machine, but not when its generated code is executed. This will ultimately result in the comparison component to report the executions as being non-equal, although the behavior of both executions is equal. During the evaluation of the case study, such differences were manually taken into account and investigated in detail. However, it has been shown that it becomes very cumbersome to manually investigate such events from the comparison result, when the number of actions of the input models is high. It would therefore be desirable to remove events from an event model captured for the fUML model, which are known to not have a corresponding match in the event model captured for the execution of the generated code and, at the same time, have no effect on the comparison of the event models.

---

accessed 10-12-2016

[3]The documentation for Java ResourceBundles is available online at `https://docs.oracle.com/javase/7/docs/api/java/util/ResourceBundle.html`, accessed 10-12-2016

[4]The documentation for AspectC is available online at `http://www.aspectc.org/doc/ac-languageref.pdf`, accessed 10-12-2016

[5]The documentation for PostSharp is available online at `http://doc.postsharp.net`, accessed 10-12-2016

Support for missing fUML Actions

One of the code generator's requirements was to provide a complete support for all fUML actions. This goal has almost been achieved. Section 4.6 lists all fUML actions that are not supported or for which only limited support is provided at this time. Section 4.6 also briefly discusses approaches for how the missing fUML actions could be supported in the future. Various fUML actions were identified for which no corresponding Java language features are available. To produce equally behaving Java code for these actions, more complex Java language constructs would have to be elaborated. However, the code generator was developed in such a way that Java mappings for missing fUML actions can be easily added so that a complete support can be provided in the future.

APPENDIX A

Implementations

All developed software artefact as well as resources designed for testing purposes are openly available in a public repository[1]. The repository consists of the following projects:

- org.modelexecution.fuml.codegen
  This package contains the developed code generation approach, the AspectJ implementations of the monitoring component and an execution component that dynamically invokes the generated code. The input fUML models are located in the *models* directory and the corresponding Java code in the *src-gen* directory.

- org.modelexecution.fuml.codegen.comparison
  This package contains the implementations of the developed custom match engine, which is used to compare two EET models.

- org.modelexecution.fuml.codegen.comparison.test
  This package contains the test suite and a component to invoke the EMF compare process and persist its results.

- org.modelexecution.fuml.codegen.eventstorage
  This package contains the elaborated EET metamodel and the storage factory which transforms the captured runtime events into EET models.

[1]The repository is available online at `https://bitbucket.org/eulbot/master-thesis`

List of Figures

List of Tables

List of Listings

Bibliography

[1] Thomas E Bell, David C Bixler, and Margaret E Dyer. An Extendable Approach to Computer-Aided Software Requirements Engineering. *IEEE Transactions on Software Engineering*, 1(1):49–60, 1977.

[2] Jean Bézivin. Model driven engineering: An emerging technical space. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 36–64. Springer, 2006.

[3] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006, ISBN: 9788131722879.

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Sngineering in Practice*. Morgan & Claypool Publishers, 2012, ISBN: 9781608458820.

[5] 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.

[6] Franck Chauvel and Jean-Marc Jézéquel. Code Generation from UML Models with Semantic Variation Points. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *LNCS*, pages 54–68. Springer, 2005.

[7] Federico Ciccozzi, Antonio Cicchetti, and Martin Sjodin. Towards Translational Execution of Action Language for Foundational UML. In *Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 153–160. IEEE, 2013.

[8] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjodin. On the Generation of Full-fledged Code from UML Profiles and ALF for Complex Systems. In *Proceedings of the 12th International Conference on Information Technology-New Generations*, pages 81–88. IEEE, 2015.

[9] Gergely Dévai, Gábor Ferenc Kovács, and Ádám An. Textual, Executable, Translatable UML. In *Proceedings of the 14th International Workshop on OCL and Textual Modeling Applications and Case Studies*, volume 1285, pages 3–12. CEUR-WS, 2014.

[10] Eclipse Projects. Xpand, 2015. Online available at: http://www.eclipse.org/modeling/m2t/?project=xpand.

[11] Gregor Engels, Roland Hücking, Stefan Sauer, and Annika Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, volume 1723 of *LNCS*, pages 473–488. Springer, 1999.

[12] Dominik Gessenharter and Martin Rauscher. Code Generation for UML 2 Activity Diagrams. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications*, volume 6698 of *LNCS*, pages 205–220. Springer, 2011.

[13] 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*, pages 16–27. ACM, 2003.

[14] 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*, volume 6394 of *LNCS*, pages 331–345. Springer, 2010.

[15] Martin Hitz and Martin Bernauer. *UML@ Work: objektorientierte Modellierung mit UML 2*. Dpunkt. verlag, 2005, ISBN: 9783898642613.

[16] IMB®. Rational Rhapsody, 2016. Online available at: www.ibm.com/software/products/de/ratirhap.

[17] IMB®. Rational Software Architect, 2016. Online available at: www-03.ibm.com/software/products/de/ratsadesigner.

[18] IMB®. Using the UML Action Language in Rational Software Architect, 2017. Online available at: goo.gl/EmdSpp.

[19] International Standards Organization. ISO 18629, Process Specification Language, 2004. Online available at: http://www.nist.gov/psl.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001.

[21] Alexander Knapp and Stephan Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In *Proceedings of the 5th Workshop on Tools for System Design and Verification*, pages 59–64. Institut für Informatik, Universität Augsburg, 2002. Online available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.8649&rep=rep1&type=pdf.

[22] Hans J. Köhler, Ulrich Nickel, Jörg Niere, and Albert Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 241–251. ACM, 2000.

[23] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *Proceedings of the International Symposium on Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 377–393. Springer, 2007.

[24] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML: An Open Source Toolset for MDA. In *Proceedings of the Tools and Consultancy Track of the Fifth European Conference on Model-Driven Architecture Foundations and Applications*, pages 1–4, 2009.

[25] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[26] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker. *Formal aspects of computing*, 11(6):637–664, 1999.

[27] Johan Lilius and Iván Porres Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of the Second International Conference on the Unified Modeling Language*, volume 1723 of *LNCS*, pages 430–444. Springer, 1999.

[28] Zhiming Liu, Jifeng He, Jing Liu, and Xiaoshan Li. Unifying views of UML. *Electronic Notes in Theoretical Computer Science*, 101:95–127, 2004.

[29] Quan Long, Zhiming Liu, Xiaoshan Li, and He Jifeng. Consistent Code Generation from UML Models. In *Proceedings of the 2005 Australian Software Engineering Conference*, pages 23–30. IEEE, 2005.

[30] Tanja Mayerhofer. *Defining Executable Modeling Languages with fUML*. PhD thesis, TU Wien, Institute of Software Technology and Interactive Systems, 2014. Online available at: `https://publik.tuwien.ac.at/files/PubDat_233990.pdf`.

[31] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fUML. In *Proceedings of the 7th Workshop on Models@run.time*, pages 53–58. ACM, 2012.

[32] Stephen J Mellor. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 2004, ISBN: 0-201-78891-8.

[33] Stephen J Mellor, Stephen Tockey, Rodolphe Arthaud, and Philippe Leblanc. An Action Language for UML: Proposal for a Precise Pxecution Pemantics. In *Proceedings of the 1st International Conference on The Unified Modeling Language: Beyond the Notation*, volume 1618 of *LNSC*, pages 307–318. Springer, 1998.

[34] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. Automatic Code Generation for Embedded Systems: From UML Specifications to VHDL Code. In *Proceedings of the 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090. IEEE, 2010.

[35] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 742–745. ACM, 2000.

[36] Jörg Niere and Albert Zündorf. Using FUJABA for the Development of Production Control Systems. In *In Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*, pages 181–191. Springer, 1999.

[37] No Magic, Inc. MagicDraw, 2016. Online available at: `www.nomagic.com/products/magicdraw.html`.

[38] OASIS. *Business Process Execution Language 2.0 (WS-BPEL 2.0)*, 2007. Online available at: `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`.

[39] Object Management Group. Unified Modeling Language (UML), 2005. Version 2.0, Online available at: `http://www.omg.org/spec/UML/2.0/`.

[40] Object Management Group. MOF Model To Text Transformation Language (MOFM2T), 2008. Version 1.0, Online available at: `http://www.omg.org/spec/MOFM2T/1.0/PDF`.

[41] Object Management Group. Unified Modeling Language (UML), Infrastructure, 2011. Version 2.4.1, Online available at: `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF`.

[42] Object Management Group. Unified Modeling Language (UML), Superstructure, 2011. Version 2.4.1, Online available at: `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`.

[43] Object Management Group. Concrete Syntax For A UML Action Language: Action Language For Foundational UML, 2013. Version 1.0.1, Online available at: `http://www.omg.org/spec/ALF/1.0.1/PDF`.

[44] Object Management Group. Unified Modeling Language (UML), 2013. Version 2.5, Online available at: `http://www.omg.org/spec/UML/2.5/PDF`.

[45] Object Management Group. Model Driven Architecture (MDA), MDA Guide, 2014. Revision 2.0, Online available at: `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf`.

[46] Object Management Group. Precise Semantics of UML Composite Structures (PSCS), 2015. Version 1.0, Online available at: http://www.omg.org/spec/PSCS/1.0/PDF.

[47] Object Management Group. Semantics of a Foundational Subset for Executable UML Models, 2015. Version 1.2, Online available at: http://www.omg.org/spec/FUML/1.2/Beta2/PDF.

[48] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[49] Douglas L Perry. *VHDL*, volume 2. McGraw-Hill, 1998, ISBN: 9780071501088.

[50] Stephan Roser, Florian Lautenbacher, and Bernhard Bauer. Generation of Workflow Code from DSMs. Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland, 2007, ISBN: 978-951-39-2915-2. Online available at: http://www.dsmforum.org/events/DSM07/papers.html.

[51] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004, ISBN: 0321245628.

[52] Stefan Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Universität Ulm, 2006. Online available at https://oparu.uni-ulm.de/xmlui/bitstream/handle/123456789/390/vts_5643_7444.pdf?sequence=1&isAllowed=y.

[53] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society*, 39(2):25–31, 2 2006.

[54] Sally Shlaer and Stephen J Mellor. *Object-oriented Systems Analysis Modeling the World in Data*. Yourdon Press, 1988, ISBN: 0-13-629023-X.

[55] Sally Shlaer and Stephen J Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1992, ISBN: 0-13-629940-7.

[56] Sparx Systems®. Enterprise Architect, 2016. Online available at: www.sparxsystems.com/products/ea.

[57] Muhammad Usman, Aamer Nadeem, and Tai-hoon Kim. UJECTOR: A Tool for Executable Code Generation from UML Models. In *Proceedings of the 2008 International Conference on Advanced Software Engineering and Its Applications*, pages 165–170. IEEE, 2008.

[58] Adriaan Van Wijngaarden. The Generative Power of Two-Level Grammars. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, volume 14 of *LNCS*, pages 9–16. Springer, 1974.

[59] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A Co-Design Approach for Embedded System Modeling and Code Generation with UML and MARTE. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 226–231. European Design and Automation Association, 2009.

[60] R Hevner von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS quarterly*, 28(1):75–105, 2004.

[61] Robert Wagner. Developing Model Transformations with FUJABA. In *Proceedings of the 4th International Fujaba Days, Technical Report tr-ri-06-275*, pages 79–82, 2006.

[62] xtUML.org. BridgePoint - Executable Translatable UML Open Source Editor, 2017. Online available at: `www.sparxsystems.com/products/ea`.

[63] Jing Yang, Quan Long, Zhiming Liu, and Xiaoshan Li. A predicative semantic Model for Integrating UML Models. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 3407 of *LNCS*, pages 170–186. Springer, 2004.