FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Breathing New Life into Models

An Interpreter-Based Approach for Executing UML Models

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Tanja Mayerhofer
Matrikelnummer 0625154

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 4. Mai 2011

(Unterschrift Verfasserin)	(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Breathing New Life into Models

An Interpreter-Based Approach for Executing UML Models

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Tanja Mayerhofer
Registration Number 0625154

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Univ.Ass. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Vienna, 4th May 2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Tanja Mayerhofer, Richtergasse 1a/5, 1070 Wien

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

Wien, 4. Mai 2011

Tanja Mayerhofer

Abstract

Over the past years *Model-Driven Development (MDD)* gained significant popularity. With the usage of this paradigm the software engineering process becomes more model-centric and less code-centric. This means that models become the main artifact in the software development process and therewith the whole software development process relies on these models and their correctness. For this reason the need for executable models that can be tested and validated arose. The de facto standard for modeling software systems is OMG's *Unified Modeling Language (UML)*. The problem is that UML models are not executable because UML has no precise and completely specified semantics. Its semantics is defined informally in English prose and this definition is scattered throughout the standard with about 1000 pages. Because of this situation, ambiguities arise and models can be interpreted and executed in different ways. This also led to the development of execution tools that are not interoperable because they implement different execution semantics.

OMG has recognized the need for executable models in an unambiguous way, and has developed a new standard called *Semantics of a Foundational Subset of Executable UML Models* or *foundational UML (fUML)* that was released in February 2011. This standard defines the precise execution semantics of a subset of UML 2, the so-called *foundational UML subset*.

The research question of this thesis is as follows. Is the semantics definition of the fUML standard sound and applicable for building tools that enable the execution of UML activity diagrams? To answer this question, a prototype of a model interpreter has been developed in this thesis that is able to execute and debug UML models according to the execution semantics defined in the fUML standard. This model interpreter prototype focuses on executing activity diagrams that model the manipulation of objects and links in a system. Furthermore, the prototype provides reasonable debugging functionality similar to the functionality offered for debugging code like the step-wise execution and the displaying of the debugging progress. The experiences gained during the implementation of the model interpreter prototype led to the following conclusion. The fUML standard is applicable for implementing tools that support the execution of UML activity diagrams, however, high efforts are necessary to develop a user-friendly and efficiently usable tool supporting features like the debugging of models or the execution of incomplete models.

Kurzfassung

Im Laufe der letzten Jahre gewann die *modellgetriebene Softwareentwicklung*, auch bekannt als *Model-Driven Development (MDD)*, enorm an Bedeutung. Dabei wird die Implementierung eines Systems, d.h. der Code, automatisch oder halbautomatisch aus den Modellen generiert. Die Korrektheit dieser Modelle ist demnach von großer Bedeutung. Damit ergibt sich die Notwendigkeit ausführbarer Modelle, die durch ihre Ausführung getestet und validiert werden können. Der objektorientiert Modellierungsstandard UML hat den Nachteil, dass er keine präzise und vollständig spezifizierte Ausführungssemantik besitzt. Die Ausführungssemantik von UML wird nur informell in englischer Prosa definiert und unstrukturiert über den 1000 Seiten umfassenden Standard hinweg verteilt behandelt. Dadurch ergeben sich Mehrdeutigkeiten bezüglich der Interpretation von Modellen, was sich auch in der Inkompatibilität der UML Werkzeuge widerspiegelt.

Die OMG erkannte das Bedürfnis nach ausführbaren Modellen sowie die Probleme der Semantikdefinition von UML und entwickelte einen neuen Standard mit dem Titel *Semantics of a Foundational Subset of Executable UML Models* oder *foundational UML (fUML)*, der im Februar 2011 in Erstversion veröffentlicht wurde. Dieser Standard definiert die präzise und vollständige Semantik einer Untermenge von UML 2, die als *foundational UML subset* bezeichnet wird.

Die Forschungsfrage dieser Arbeit lautet daher: Ist die Semantikdefinition des fUML Standards geeignet, um Programme zu implementieren, die das Ausführen von UML Aktivitätsdiagrammen ermöglichen? Um diese Frage zu beantworten, wurde ein Prototyp für einen Modell-Interpreter entwickelt, der UML Modelle entsprechend der im fUML Standard definierten Ausführungssemantik ausführen und debuggen kann. Dieser Modell-Interpreter konzentriert sich dabei auf die Ausführung von UML Aktivitätsdiagramme, die sich mit der Manipulation von Objekten und Links in einem System beschäftigen. Weiters stellt der Prototyp sinnvolle Debugging-Funktionen zur Verfügung, ähnlich jener Funktionalitäten, die vom Debuggen von Code bekannt sind, wie beispielsweise die schrittweise Ausführung oder das Anzeigen des Debugging-Fortschritts. Die Erfahrungen, die im Zuge der Implementierung dieses Prototyps gewonnen werden konnten zeigen, dass der fUML Standard verwendet werden kann, um Programme zu entwickeln, die das Ausführen von UML Aktivitätsdiagrammen ermöglichen. Gleichzeitig ist aber ein hoher Implementierungsaufwand nötig, um benutzerfreundliche und effizient nutzbare Werkzeuge zu entwickeln, die Funktionalitäten wie das Debuggen von Modellen oder das Ausführen unvollständiger Modelle unterstützen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The Unified Modeling Language[1] (UML) is the widely accepted standard for modeling software systems. It is a graphical modeling language that is used to specify, construct, visualize and document a software system [25] and it was developed and standardized by the Object Management Group[2] (OMG). Structural as well as behavioral aspects of a software system are specified throughout the development process in UML models which are refined in the various consecutive steps of the development lifecycle from requirements analysis to maintenance. As correcting an error in a system becomes more expensive the later the error is detected, it is very important to discover and correct errors in a very early stage of the software development process. And because models are often used as specification for the implementation of a system, it is essential to detect errors in these models before they are reflected in the source code. But at the moment these models can only be read and manually reviewed. There is no way to test models until the executable code is available, because models are scarcely executable in modeling tools.

Another reason why it is desirable to have executable models is that with the development paradigm Model-Driven Development (MDD) the software development process becomes more model-centric and less code-centric. This shift implicates that models are no longer only used to document design decisions and to support a thorough understanding of the system, but models are the primary artifact of the software development process and the code is automatically generated from them. This means that the development of a software system is model-driven instead of only model-based. OMG's MDD approach which is known as Model Driven Architecture[3] (MDA) suggests the usage of UML to define a platform-independent model (PIM) of the system under development. This model considers the entire system but omits the details of

[1]http://www.omg.org/spec/UML
[2]http://www.omg.org
[3]http://www.omg.org/mda

the implementation platform. The second step is to transform the platform-independent model into a platform-specific model (PSM) that adds the details of a certain implementation platform. The last step is then to generate source code automatically or semi-automatically out of these models. Because of this shift to a model-centric software development process, the need for executable models that can be debugged and executed becomes more evident.

1.2 Problem Statement

It is desirable to have models that can be executed and debugged. The main reason for this is that models can be tested thoroughly and so errors can be detected more easily with a tool that enables the execution and debugging of models. Also the understanding of the models, and therewith the understanding of systems, could be improved if models can be debugged. MDA is an ongoing trend in software engineering that makes the models of a system to the main artifact of the software development process. This trend makes the need for executable models even more obvious.

The problem with the usage of UML to model a system, like it is suggested by MDA, is that UML has no precise and complete specified execution semantics. But to define executable models, a defined semantics is essential. UML's execution semantics is only informally defined in English prose and it is scattered throughout the standard. This leads to lots of problems. Because the semantics definition is much dispersed in the standard, it is very difficult to get a global understanding of UML's semantics and another consequence is that there are logical inconsistencies and omissions in the semantics definition. Because the semantics definition of UML is neither precise nor complete, ambiguities arise and models can be misinterpreted. But to execute UML models their execution semantics has to be clear. Here the question arises what benefits a standard has when models are interpreted and executed differently by different people. This problem also led to the development of tools for executing UML models that implement different execution semantics.

Lots of publications exist that are concerned with the missing formal semantics specification of UML. The precise UML Group [7] and the UML Semantics Project [2] have to be named in this context. But currently no semantics definition for the whole UML standard exists.

1.3 Aim of the Work

The aim of this master thesis is to build a prototype of a model interpreter for UML models. This model interpreter prototype shall focus on executing activity diagrams because one fundamental principle of UML's semantics is that every behavior in a system is eventually caused by actions and therewith every kind of behavior in UML is expressible as a sequence of actions. To be precise the prototype shall enable the user to execute and debug activity diagrams that model the manipulation of objects and links. The interpreter shall execute activity diagrams that define, based on a class diagram, how objects are created and destroyed, how links between objects are created and destroyed, how attribute values of objects are set and removed etc. So only selected modeling concepts of UML activity diagrams shall be supported by the model interpreter.

As mentioned before the execution of models requires a precise definition of the execution semantics of these models. The "Semantics of a Foundational Subset for Executable UML Models", or short foundational UML[4] (fUML), is a new standard of the OMG that precisely defines the execution semantics for a subset of UML 2.3, whereat the version 1.0 only supports activity diagrams. Because fUML is an OMG standard defining the execution semantics of OMG's UML, this standard is most promising to be highly accepted in the UML community. Out of this consideration the semantics definition of activity diagrams defined in fUML was chosen to build the prototypical model interpreter.

The model interpreter prototype is implemented using the Eclipse Modeling Framework[5] (EMF) because this framework ideally supports all functionality needed to build it: It supports the definition of metamodels, the code generation from these metamodels and the automatic generation of editors that can be used to instantiate models from the metamodels. The model interpreter prototype is implemented as Eclipse plug-in to ease its provision.

1.4 Methodological Approach

The methodological approach for this master thesis consists of three parts:

1. **Literature survey**. As the aim of the work is to build a prototypical model interpreter for activity diagrams, the first step consists of a literature survey for a semantics definition of UML activity diagrams. This survey lead to the new fUML standard of OMG that was available as Beta 3 specification at the start of this thesis.

2. **Implementation**. Based on the execution semantics of UML activity diagrams which is defined in the fUML standard, the implementation of an Eclipse plug-in based on EMF for executing and debugging selected concepts of activity diagrams build-up the second step.

3. **Evaluation**. As the last step, the implementation of the model interpreter prototype for UML activity diagrams is evaluated and possible enhancements are identified. Also experiences with the fUML standard are reflected and limitations are presented.

[4]http://www.omg.org/spec/FUML
[5]http://www.eclipse.org/emf

1.5 Structure of the Work

This thesis consists of further six chapters.

Chapter 2 and Chapter 3 are concerned with the theoretical background of this thesis, i.e., with UML and fUML.

Chapter 2 gives an introduction into UML and presents an overview of the thirteen diagram types of UML. Because this thesis deals with the execution of activity diagrams, this diagram type is described in more detail. The UML metamodel, i.e., the syntax of UML, is presented as well as the UML semantics architecture.

The new fUML Standard of OMG is outlined in Chapter 3. Its objectives and structure are presented. This chapter compares the modeling concepts supported by fUML with the modeling concepts offered by UML. fUML defines the execution semantics of a foundational subset of UML and an execution engine for executing models based on this foundational subset. Chapter 3 also deals with this execution engine.

Chapter 4 is concerned with the implemented Eclipse plug-in for executing and debugging activity diagrams. Here the supported modeling concepts for activity diagrams are presented in more detail and the functionality of this prototypical model interpreter is treated.

Chapter 5 discusses the experiences concerning fUML that were gained during the implementation of the Eclipse plug-in.

Related work is introduced in Chapter 6 and the similarities and differences compared to the work on hand are pointed out.

Chapter 7 provides a summary that sums up the main points of this thesis.

CHAPTER 2

Unified Modeling Language (UML)

2.1 Introduction to UML

The Unified Modeling Language (UML) is an object-oriented graphical modeling language which was developed and standardized by the Object Management Group (OMG). This standard, which is currently in version 2.3[1], intends to incorporate experience about modeling techniques as well as best practices in software development. UML is used to specify, construct, visualize and document the artifacts of a software system and it is a widely accepted standard for modeling software systems [25].

The following characteristics are the reason why the UML is named **Unified** Modeling Language [25]:

- **Historical methods and notations**. As stated before UML incorporates experiences about modeling techniques and best practices in software development.
- **Application domain**. UML is a general-purpose modeling language. It can be used for a wide range of application domains. For instance distributed systems can be modeled as well as real-time system.
- **Development lifecycle**. UML can be used throughout the software engineering process from requirements analysis to maintenance. This is important because models of a software system are naturally not created all at once. They are created and refined successively during the development process.
- **Development process**. UML can be used as modeling language independent from the used development process. Particularly it supports iterative and incremental processes.

[1]http://www.omg.org/spec/UML/2.3

- **Tools, languages, and platforms**. UML can be used independent from the used development tools, implementation platform, and programming language.

- **Internal concepts**. The modeling concepts of UML are defined in a metamodel. This metamodel was constructed under to consideration of enabling a broad applicability of UML.

UML 2 consists of four parts, namely of the infrastructure, the superstructure, the object constraint language (OCL) and the diagram interchange [10].

- **Infrastructure**. The infrastructure forms the basis for the language definition of UML 2 and therewith of the superstructure.

- **Superstructure**. The superstructure is the actual definition of the UML 2 modeling language.

- **Object constraint language**. OCL is a constraint language that can be used to define constraints on the models as well as to formulate queries on models.

- **Diagram interchange**. The diagram interchange is concerned with the exchange of layout information of a UML diagram to facilitate the diagram interchange between UML tools.

2.2 Overview of UML Diagram Types

The UML consists of thirteen diagram types. Each type describes another view on the modeled software system. The diagram types can be categorized into diagrams for *modeling the structure* of a system and diagrams for *modeling the behavior* of a system. This categorization is displayed in Figure 2.1.

Class diagram, object diagram, package diagram, component diagram, composite structure diagram and deployment diagram are the six diagram types used for molding structure whereas use case diagram, activity diagram, state machine, sequence diagram, communication diagram, timing diagram and interaction overview diagram are the seven diagram types for modeling the behavior of a system. The last four of these diagram types (sequence, communication, timing and interaction overview diagram) can be grouped into the category of *interaction diagrams* because they are all concerned with the interactions between objects of a system.

These thirteen diagram types of UML are briefly described in the following sections (cf. [10]).

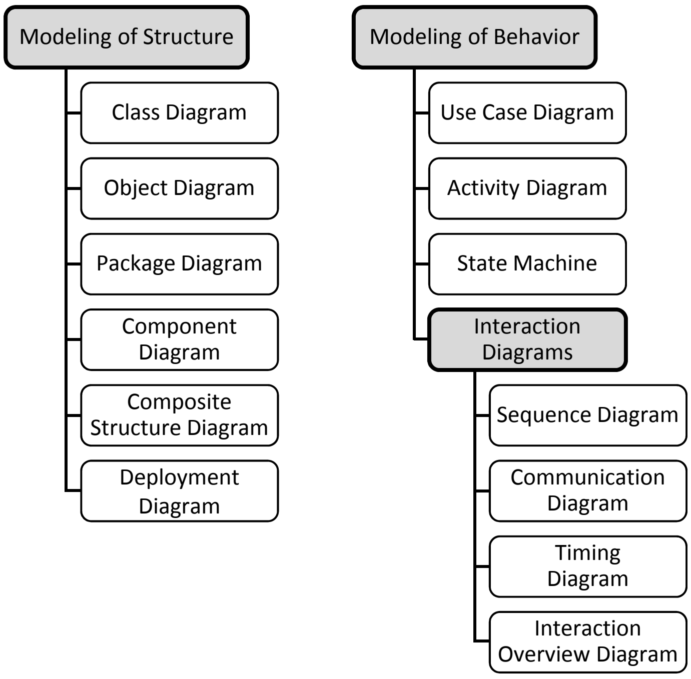


Figure 2.1: Categorization of the UML diagram types

Modeling of Structure

Class diagram. The class diagram is used to describe the classes of a software system, their attributes, operations and relationships among each other.

Object diagram. To describe an exemplary system configuration in terms of existing objects, their attribute values and relationships to other objects, the object diagram is used.

Package diagram. The package diagram is used to group elements of a system into packages which can have relationships among each other. Thereby it is a means to structure the elements of a system.

Component diagram. The component diagram allows modeling the components of a system and their dependencies. A component is a modular part of the system and provides defined interfaces.

Composite structure diagram. To hierarchically decompose elements of the modeled system like classes, one can make use of the composite structure diagram.

Deployment diagram. To model the runtime systems as well as the communication between its elements, the deployment diagram is used. It also enables the specification of the deployment of run-time artifacts like files, on the resources of a runtime system like a server.

Modeling of Behavior

Use case diagram. The use case diagram is used to describe the functionality which the modeled software system provides the user. The functionality is modeled in terms of use cases. The interaction of the users or actors with the system's functionality or use cases is modeled as well as the relationships between the actors and the relationships between the use cases.

Activity diagram. To model how actions in the system are executed, the activity diagram can be used. It describes the control as well as the data flow between the actions of a system.

State machine. A state machine describes the lifecycle of an object in terms of the states of an object, the possible state transitions and the actions which can be executed in each state.

Sequence diagram. The sequence diagram is used to model the interactions between objects which are necessary to accomplish a given task. The main focus of this diagram is to specify the chronology of the interactions.

Communication diagram. Like the sequence diagram also the communication diagram is used to model the interactions between objects. But here the focus lies on the structural relationships between the interaction partners.

Timing diagram. The timing diagram is also concerned with the interactions between objects. It allows to explicitly specify the state changes of the interacting objects during the interaction.

Interaction overview diagram. The interaction overview diagram is used to visualize in which order interactions can take place. It therewith describes the coordination of the different interactions.

2.3 Modeling Behavior using Activity Diagrams

Because this thesis is concerned with the execution of activity diagrams, this UML diagram type is introduced in more detail.

As stated before, activity diagrams are used to describe the behavior of a system. It is concerned with the description of the steps necessary to accomplish a given task. The activity diagram can be used to describe workflows at a very highly level of abstraction as well as to describe the instructions necessary in an operation of a class at a very low level of abstraction. Because this thesis deals with executing activity diagrams, it is necessary to describe activities in a very detailed way, i.e., executable activities have to be described at a very low level of abstraction.

Figure 2.2 shows an excerpt of the metamodel that contains the basic concepts for modeling activities. The UML metamodel is described in more detail in Chapter 2.4. The notations of the modeling concepts for activities and further details are presented in Table 2.1.

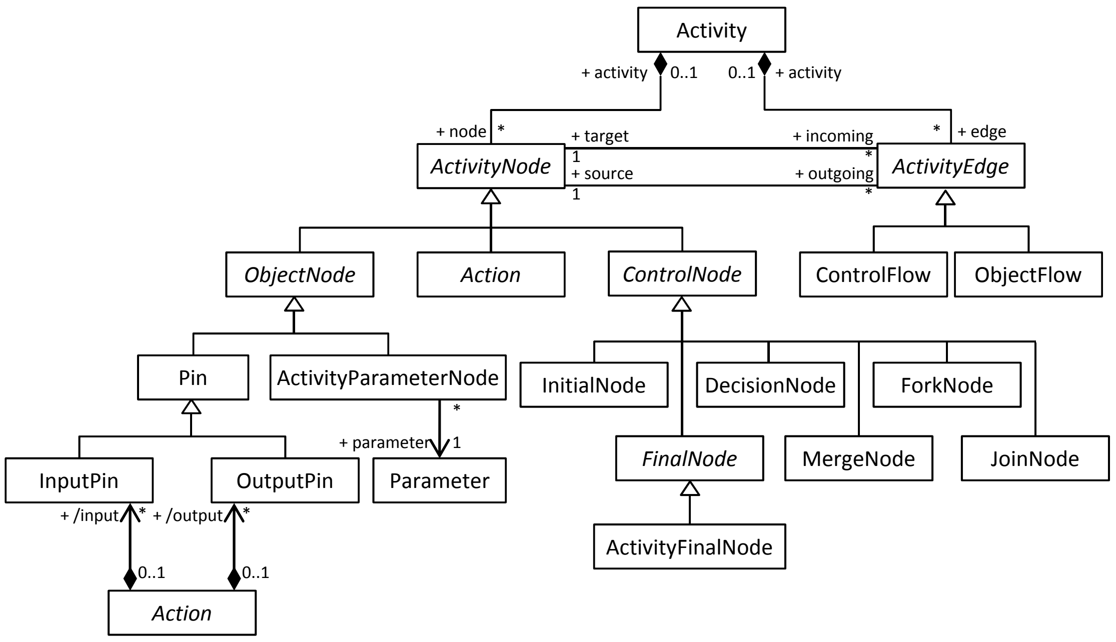


Figure 2.2: Excerpt of the UML metamodel that contains the basic concepts of activity diagrams [19]

Activities consist of *activity nodes* and *activity edges*. *Actions* are activity nodes that define the single steps of an activity. Actions can also process data and therefore have inputs and outputs which are modeled using so-called *input pins* and *output pins*. Also an activity can have inputs and outputs which are specified by *activity parameter nodes*. Pins and activity parameter nodes are *object nodes*. To define the start of an activity, the end of an activity, alternative branches or concurrent branches, *control nodes* are used. The *initial node* defines the starting points of an activity, whereas the *activity final node* determines the end of an activity. Alternative branches are modeled using a *decision node* that defines under what condition which of the branches is executed. Alternative branches are merged using the *merge node*. With the *fork node* concurrent actions can be defined which can again be synchronized using the *join node*. Activity nodes are connected by activity edges. *Control flow* edges are used to define the control flow among activity nodes whereas *data flow* edges are used to model the data flow.

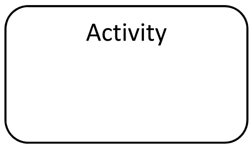
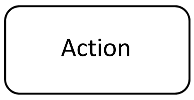
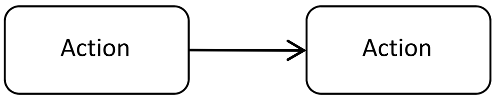
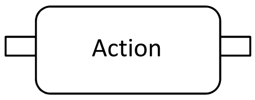
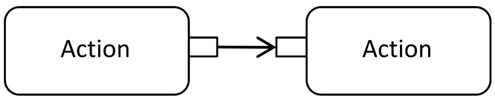
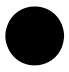
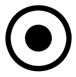
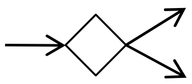
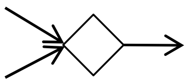
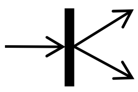
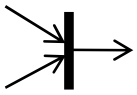
Modeling concept	*Notation*	*Description*
Activity		Activities describe the behavior of a system.
Action		Actions represent the individual steps necessary to accomplish an activity.
Control flow		A control flow edges describe the control flow through the actions.
Action with input and output pins		Actions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

Table 2.1: Modeling concepts of UML activity diagrams

Another important concept for understanding activity diagrams is the *token concept* that originates from Petri Nets. *Tokens* are no additional modeling concept, so there exists no notation for them and they are also not included in the metamodel of UML. Tokens are used as a coordination mechanism to describe possible execution flows of an activity. So tokens represent control and they can also carry data. Tokens which carry data are called *data tokens* or *object tokens*, tokens that do not carry information are called *control tokens*. The tokens flow along activity edges from one activity node to another. An activity node can be executed if there are tokens present at all incoming edges. After the execution tokens are provided on all outgoing edges and therewith the execution of subsequent activity nodes may be triggered. There can be more than one token present in an activity diagram during execution, e.g., when fork nodes start concurrent execution flows or if more than one initial node are present.

UML provides predefined *primitive actions* for modeling the manipulation of objects and links, computation and communication among objects. [25] groups these actions into ten categories. This categorization is presented in Table 2.2. These actions are particularly important for the execution of activity diagrams because they are primitive enough to be interpreted and executed by a computer.

Category	*Action*	*Purpose*
classification	readIsClassifiedObject	test classification
	reclassifyObject	change classification
	testIdentity	test object identity
communication	broadcastSignal	broadcast
	callOperation	normal call
	reply	reply after explicit accept
	(implicit) return	implicit action on activity end
	sendObject	send signal as object
	sendSignal	send signal as argument list
computation	acceptCall	inline wait for call
	acceptEvent	inline wait for event
	addVariableValue	add additional value to set
	applyFunction	mathematical computation
	callBehavior	invoke behavior
	clearVariable	reset value in procedure
	readSelf	obtain owning object identity
	readVariable	obtain value in procedure
	removeVariableValue	remove value from set
	writeVariable	set value in procedure
control	startOwnedBehavior	explicit control
creation	createLinkObject	create object from association
	createObject	create normal object
destruction	destroyObject	destroy object
exception	raiseException	raise exception in procedure

Category	*Action*	*Purpose*
read	readExtent	get all objects
	readLink	get link value
	readLinkObjectEnd	get value from association class
	readLinkObjectEndQualifier	get qualifier value
	readStructuralFeature	get attribute value
time	durationObservation	measure time interval
	timeObservation	get current time
write	addStructuralFeatureValue	set attribute value
	clearAssociation	clear links
	clearStructuralFeature	clear attribute value
	createLink	add a link
	destroyLink	remove a link
	removeStructuralFeatureValue	remove value from set

Table 2.2: Primitive actions of UML [25]

2.4 Syntax of UML - The UML Metamodel

The UML modeling language is defined in the UML superstructure. The abstract syntax of the UML modeling concepts is specified by means of a metamodel. Also the concrete syntax of the concepts, i.e., their notation, is defined by the UML superstructure but is not part of the metamodel.

OMG Four-Layer Metamodel Hierarchy

A *metamodel* is the specification of a language which is used to construct models. It describes which modeling concepts exists, which attributes they have, which relationships between them can exist etc. This is the *abstract syntax* of a modeling language. A model is an instance of a metamodel, i.e., each element of the model is an instance of an element of the metamodel. This structure can be recursively applied so that a model can also be again a metamodel and therefore a language specification. This recursive application leads to a *metamodel hierarchy*. Modeling language specifications of the OMG are developed within the framework of a four-layer metamodel hierarchy which is depicted in Figure 2.3 [18].

Meta-metamodel. The *meta-metamodel layer*, also called M3, constitutes the basis of the metamodel hierarchy. It is responsible for specifying the language used to define a metamodel. In the four-layer metamodel hierarchy the Meta Object Facility[2] (MOF) serves as meta-metamodel.

[2]http://www.omg.org/mof

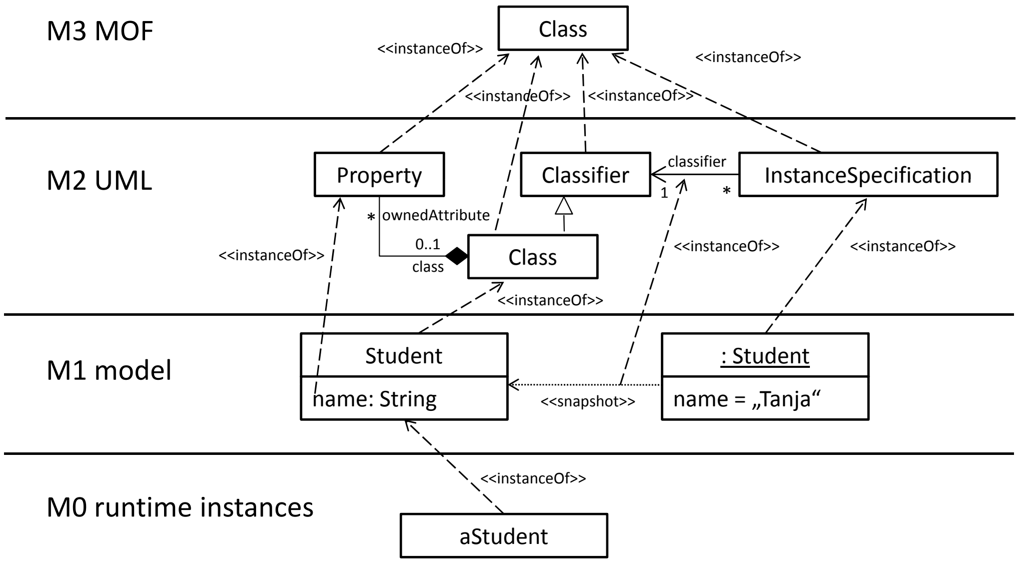


Figure 2.3: Four-layer metamodel hierarchy [18]

Metamodel. The *metamodel layer* is also referred to as M2. A metamodel is an instance of a meta-metamodel and it is used to specify the language which is used to construct models. In the case of UML the UML metamodel is placed on layer M2, it is an instance of MOF which is its meta-metamodel. Figure 2.3 presents parts of the abstract syntax of the class diagram and of the object diagram: A class is a classifier that owns an arbitrary number of properties and an instance specification (this modeling concept represents the objects in a system) has exactly one classifier. Another excerpt of the UML metamodel that deals with the abstract syntax of activity diagrams was already presented, namely in Figure 2.2 of Chapter 2.3.

Model. The so-called M1 layer is concerned with the *models* which are instances of the metamodel. A UML model is an instance of the UML metamodel. Figure 2.3 shows elements of a model that conform to the abstract syntax defined on layer M2: The model contains a class called “Student” that has a property called “name” and an object of that class with the value “Tanja” for the property. This example makes clear that an instantiation relationship does not only exist between the UML metamodel on layer M2 and the model on layer M1 but also within the model layer M1. The former case is called a *linguistic instantiation relationship* and the latter is called an *ontological instantiation relationship* [10]. So Figure 2.3 shows a linguistic instantiation relationship between the object of the class “Student” on layer M1 and the metaclass instance specification on M2 (“instance of”) and an ontological relationship between the object of the class “Student” and the class “Student” (“snapshot”) which are both situated on layer M1.

Runtime instances. The lowest layer M0 contains the *runtime instances* of the model elements of the model layer M1. Figure 2.3 shows the runtime instance “aStudent” which is an instance of the class “Student”.

Structure of the UML Metamodel

The UML superstructure, i.e., the UML metamodel, is structured modularly. This is because UML can be used in a wide range of application domains and not all application domains need every modeling concept provided by UML. To enable that users only select the needed parts of the UML, it is structured into so-called language units. The structuring was carried out under the consideration of the interchangeability of UML models to support the interoperability of UML tools. For this reason also compliance levels and syntax compliance were introduced. I.e., the UML metamodel was structured using the three criteria language unit, compliance level and syntax compliance [19].

Language units. Language units encapsulate associated modeling concepts. Such language units are the state machine language unit and the activities language unit.

Compliance levels. Further the UML metamodel is structured into four compliance levels. Starting with level L0 (foundation level) each level adds further modeling concepts and there-with enlarges the modeling capabilities. The further levels are L1 (basic level), L2 (intermediate level) and L3 (complete level).

Syntax compliance. The last criteria for structuring UML is the syntax compliance. It is distinguished between abstract syntax compliance and concrete syntax compliance. The abstract syntax is as already mentioned defined in the metamodel itself whereas the concrete syntax is defined separately.

Figure 2.4 depicts the language units of UML. The green colored language units are used to model the structure of a system and the red colored are used for modeling the behavior of a system. For convenience the dependencies between these language units are not displayed. Figure 2.4 also displays the sub-packages of the actions language unit contained on the compliance levels L1-L3.

2.5 Semantics of UML

The abstract syntax of UML is precisely and completely specified in the UML metamodel. Also the concrete syntax of most of the modeling concepts is clearly defined in the standard. But the execution semantics of UML is neither precisely nor completely specified. The execution semantics is only informally defined in English prose and it is scattered throughout the standard.

UML Semantics Architecture

The UML standard only provides a very high-level view of the run-time semantics of UML in the beginning of the standard document. The detailed semantics descriptions are then covered

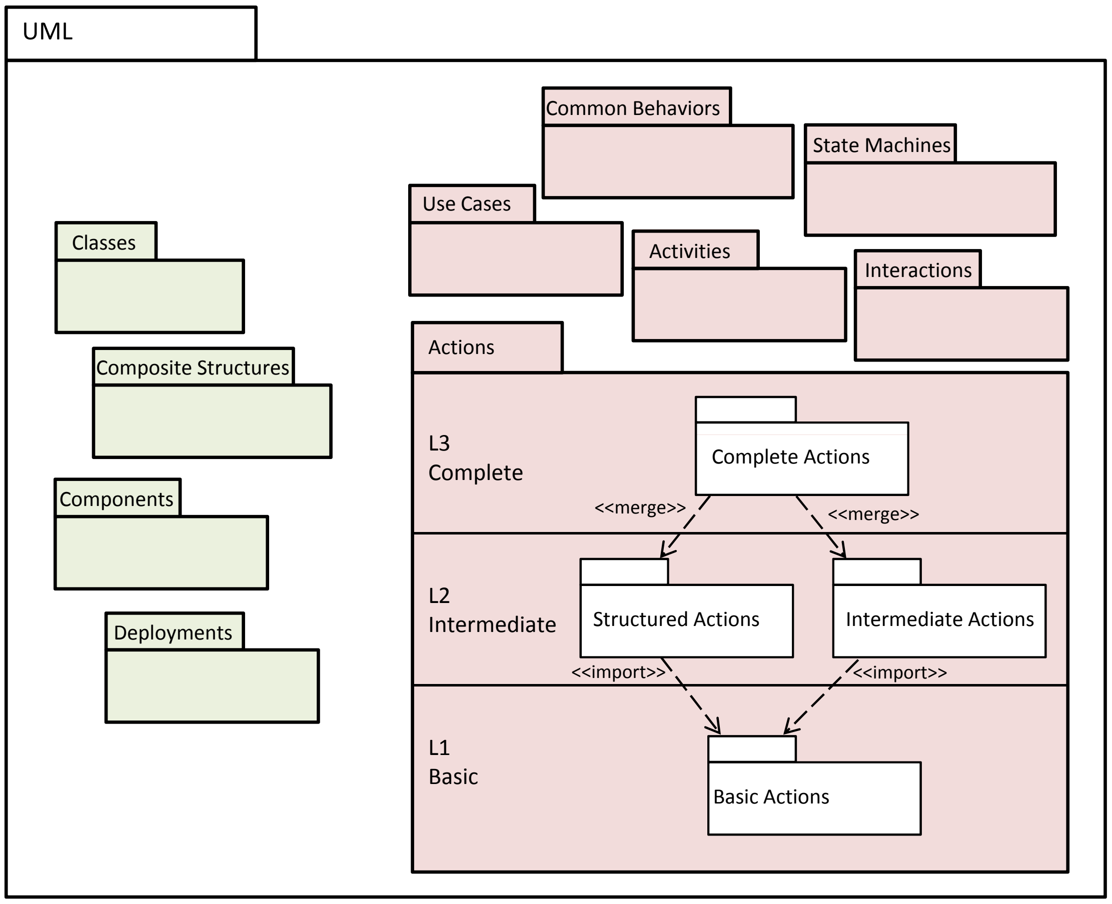


Figure 2.4: Compliance levels of UML

in the descriptions of the individual modeling concepts. Because of this much dispersed semantics specification it is very difficult to get a global understanding of UML's semantics. Another consequence is that there are logical inconsistencies and omissions in the definition of the UML semantics.

The UML standard premises two fundamental principles to describe the nature of UML's semantics [19]:

1. Every behavior in a system is eventually caused by actions which are executed by an active object.
2. The UML behavioral semantics only deals with discrete, i.e., event-driven behaviors.

Figure 2.5 depicts the semantic areas of the UML standard. The semantic areas are organized in three layers where each layer depends on the underlying layers.

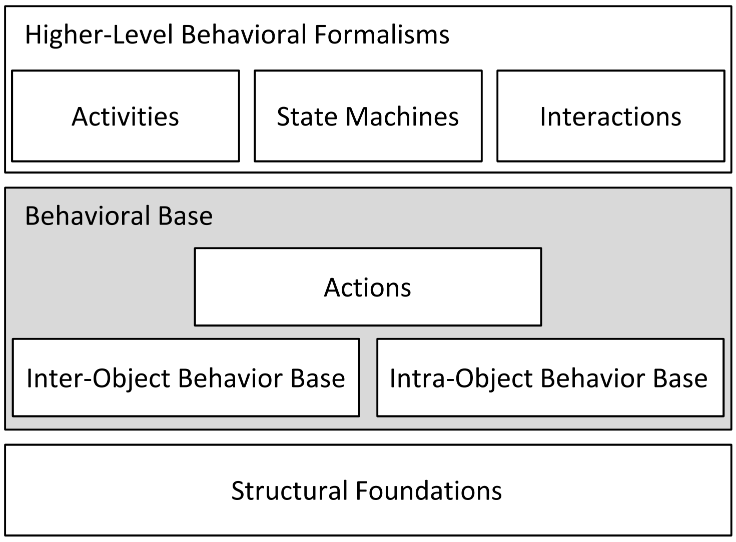

Higher-Level Behavioral Formalisms
Activities
State Machines
Interactions
Behavioral Base
Actions
Inter-Object Behavior Base
Intra-Object Behavior Base
Structural Foundations

Figure 2.5: Semantic areas of UML [19]

Structural foundations. The structural foundations form the basis of this hierarchy. This layer is concerned with structural entities like objects, links, messages etc. Therewith it copes with the first premise of UML's semantics that every behavior in a system is caused by an active object, i.e., a structural element.

Behavioral base. The next layer is the so-called behavioral base and it is concerned with the semantics of the behavior of a system. This layer is subdivided into two layers. The bottom layer consists of the *inter-object behavior base* and the *intra-object behavior base* which are concerned with the inter- and the intra-object behavior, i.e., the communication between and the behavior within structural entities. On top of this two layers the *actions* layer is placed. It deals with the semantics of individual actions which are the fundamental units of behavior.

Higher-level behavioral formalisms. The layer on the top specifies the semantics of the higher-level formalisms of UML that deals with the behavior of a system, namely *activities*, *state machines* and *interactions*.

One publication that is concerned with the UML semantics architecture is [26]. It provides a high-level view of the run-time semantics underlying UML 2.0 and describes the two lower semantic layers structural foundations and behavioral base in more detail, but it does not cover activities.

Problems of the Missing Formal Semantics Specification

Because UML is only informally specified in English prose and this specification is neither precise nor complete, ambiguities arise. This can lead to serious problems [7].

Without a clear semantics definition it is more difficult to learn and use a language, because a language's semantics is essential for understanding its usage. Since the UML standard leaves much room for interpretation regarding its semantics, users of UML can develop different variants of a precise semantics for UML on their own, which may be inconsistent. Such more precise semantics interpretations of UML are based upon the experiences of the individual users and if they are not explicitly documented this again leads to misinterpretations and communication problems.

Because the semantics definition of UML lets room for interpretation, UML models can be misinterpreted, leading to situations where the model of a system and the implementation of the system are inconsistent, because the software architect and the software engineer may have interpreted the models in different ways. In such a case the interpretations of a model have to be communicated explicitly, what can lead to additional time and effort that has to be spent in the development process.

The ambiguous and imprecise semantics definition also leads to complications in developing tools with UML support and the interoperability of UML tools is affected because every tool vendor may implement another variant of a precise semantics. Interoperability-problems can also arise of the fact that the semantics definitions of the modeling concepts also include so-called semantic variation points which leave space to refine the general UML semantics for a special domain. Also these semantic variation points are scattered throughout the standard and no official list of them exists which makes it more difficult to cope with them.

Without precise semantics definition it is also hard to validate or verify UML models formally. Mostly they are validated and verified informally for instance in manual reviews or inspections.

Also for executing UML models a complete and precise semantics is essential.

Example for Ambiguity in Semantics Definition

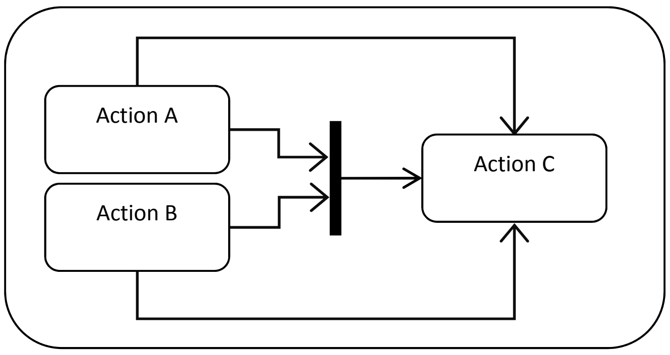


Figure 2.6: Example for ambiguities in UML semantics

The model in Figure 2.6 is used to present an example for ambiguities in the UML semantics definition. The activity has three actions called action A, action B and action C. After starting the activity, action A and action B are executed concurrently. They both have two outgoing edges each. One outgoing edge leads to action C and the other to a join node which itself again leads to action C. The question is how often action C will be executed. Is it executed never, once or twice? To answer this question statements of the UML standard version 2.3 are presented.

> "Except where noted, an action can only begin execution when it has been offered control tokens on all incoming control flows and all its input pins have been offered object tokens sufficient for their multiplicity." [19, p. 320]

This is an excerpt of the semantics definition of actions. It defines that an action can only be executed if the necessary control and object tokens are present on every incoming edge.

> "When completed, an action execution offers any object tokens that have been placed on its output pins and control tokens on all its outgoing control flows (implicit fork), and it terminates. (...) The offered tokens may now satisfy the control or object flow prerequisites for other action executions." [19, p. 320]

Also this excerpts stems from the semantics definition of actions. Because of this definition we conclude that an action places tokens on all outgoing edges after finishing execution. Based on this interpretation the control flow in the activity presented in Figure 2.6 would look like follows: After action A finishes execution it places one control token on each of the two outgoing edges. Also action B places tokens on each outgoing edge after execution. Because now only two of the three incoming edges of action C have tokens, action C can't be executed. But both of the incoming edges of the merge node have tokens so this control node is executed.

According to the semantics definition of the join node in the standard, the control flow is continued as follows:

> "If there is a token offered on all incoming edges (of a join node), then tokens are offered on the outgoing edge according to the following join rules: 1. If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge. (...)" [19, p. 394]

So now there are control tokens on each incoming edge of action C and action C is executed.

But the following excerpt of the semantics definition of an activity leads to an ambiguity:

> "Since multiple edges can leave the same node, the same token can be offered to multiple targets. However, a token can only be accepted at one target. This means flow semantics is highly distributed and subject to timing issues and race conditions,

as is any distributed system. (...) It is the responsibility of the modeler to ensure that timing issues do not affect system goals, or that they are eliminated from the model." [19, p. 327]

This excerpt leads to the conclusion that after action A (or action B respectively) is executed either the join node or action C can accept the offered token but not both. In this interpretation, action C is never executed because only the following four possible scenarios exist:

1. After action A is executed it offers one token to the two targets action C and the join node. The token is accepted by action C. Action C can't be executed because only one of the three incoming edges provides a token. After action B is executed it offers one token to the two targets action C and the join node. The token is accepted by action C. Action C can't be executed because only two of the three incoming edges provide a token. No further action can be executed.

2. After action A is executed it offers one token to the two targets action C and the join node. The token is accepted by action C. Action C can't be executed because only one of the three incoming edges provides a token. After action B is executed it offers one token to the two targets action C and the join node. The token is accepted by the join node. Also the join node can't be executed because only one of the two incoming edges provides a token.

3. After action A is executed it offers one token to the two targets action C and the join node. The token is accepted by the join node. The join node can't be executed because only one of the two incoming edges provides a token. After action B is executed it offers one token to the two targets action C and the join node. The token is accepted by action C. Also action C can't be executed because only one of the three incoming edges provides a token.

4. After action A is executed it offers one token to the two targets action C and the join node. The token is accepted by the join node. The join node can't be executed because only one of the two incoming edges provides a token. After action B is executed it offers one token to the two targets action C and the join node. The token is accepted by the join node. Now the join node can be executed and it offers one token to action C but action C can't be executed because only one of the three incoming edges provides a token.

If UML would have a precise semantics definition, ambiguities like this would not arise.

The missing formal specification causes problems for executing UML models because a precise and complete execution semantics is missing.

CHAPTER 3

Semantics of a Foundational Subset for Executable UML Models (fUML)

3.1 Introduction to fUML

The “Semantics of a Foundational Subset for Executable UML Models” is a new standard of the OMG that defines a precise execution semantics for a defined subset of UML 2.3, the so-called *foundational UML (fUML)*. This thesis is based on the Beta 3 specification which is the finalized specification for fUML. The version 1.0 was released in the end of February 2011. OMG provides the formal specification as well as the metamodel of fUML[1].

The fUML standard deals with the two lower layers of the semantic areas of the UML 2 standard which is depicted in Figure 2.5 of Chapter 2.5. These two layers are the structural foundations and the behavioral base. The behavioral base itself consists of the inter-object behavior base, the intra-object behavior base and actions. Currently only activity diagrams are supported by fUML [20].

Foundational UML Subset

The selected subset of the UML 2 metamodel, that comprises the foundational UML, constitutes the foundation for eventually defining the execution semantics of the higher-level UML modeling concepts. This is visualized in Figure 3.1. The so-called *surface UML subset* is the subset of UML which is used to model a system. This surface UML subset typically contains more modeling concepts than the *foundational UML subset*. Because of this a translation from the surface UML subset to the foundational UML subset has to be carried out. Then the model, now being defined by means of fUML modeling concepts, can be further translated into the *computational*

[1]http://www.omg.org/spec/FUML/1.0

platform language used to execute the model. Thus fUML can be seen as an intermediary between the surface UML subset for modeling a system and the computational platform language used for executing this model of a system. fUML is a computationally complete language for executable models. This means that this subset of UML is expressive enough to enable the creation of models that can be automatically executed [20].

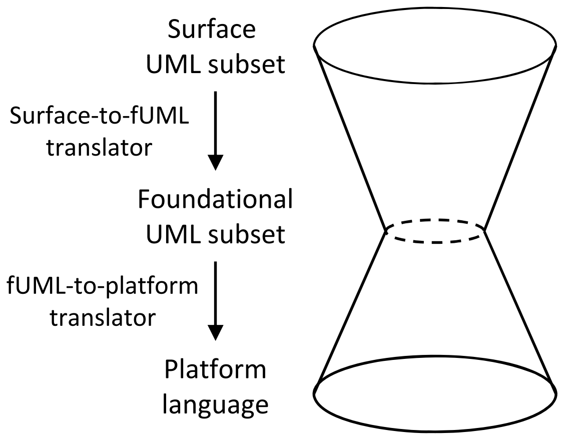


Figure 3.1: fUML as intermediary between a surface UML subset and a computational platform language [20]

The foundational subset was chosen under consideration of three criteria: compactness, ease of translation and action functionality. *Compactness* in this context means that the foundational UML subset must be small enough to precisely define its semantics. With *ease of translation* is meant that the surface-to-fUML as well as the fUML-to-platform translation should be easy. *Action functionality* means that the semantics description of fUML only defines how UML actions are executed using primitive functionality.

Of course these criteria affect each other and by specifying the foundational UML subset a trade-off between them had to be taken into consideration. For instance there is a conflict between compactness and ease of translation: If the translation could be simplified because a modeling concept in the surface UML subset has a one-to-one mapping into the computational platform language the inclusion of this modeling concept into fUML would cause an impairment of compactness.

fUML Execution Semantics

The fUML standard provides a precise definition of the execution semantics of the individual modeling concepts in the foundational UML subset using Java code and it defines a basic virtual machine for executing UML models.

Conformance in fUML not only means syntactic conformance like in UML but also semantic conformance. *Syntactic conformance* means that conforming models comply with the metamodel, i.e., the abstract syntax of fUML. *Semantic conformance* denotes that models have to be executed in the way it is defined in the semantics specification of fUML. fUML is like UML also structured into the compliance Levels L1, L2 and L3. So syntactic conformance and semantic conformance have to be seen relative to these compliance levels, i.e., the conformance to a specific fUML compliance level induces syntactic as well as semantic conformance.

3.2 Syntax of fUML - The Modeling Concepts of fUML

As stated before the fUML standard defines a subset of the UML modeling concepts and defines a precise execution semantics for the selected elements. Because fUML is just a subset of UML, i.e., the metamodel of fUML is a subset of the metamodel of UML, it is structured in the same way as UML. This means that the metamodel is grouped into language units and compliance levels and that the package structure of the fUML metamodel is the same as the package structure of UML. Packages that are not included in fUML are entirely excluded. Packages that are included may be restricted compared to the corresponding package in UML, i.e., some elements of the package may be excluded and additional constraints may be defined. Also the four-layer metamodel hierarchy is applicable for fUML just as for UML.

Table 3.1 depicts which of the packages of the UML metamodel are included in fUML, whereat we distinguish between packages for structural modeling and packages for behavioral modeling.

UML package	*Included in fUML?*
Modeling of structure	
Classes	yes
Components	no
Composite Structures	no
Deployments	no
Modeling of behavior	
Actions	yes
Activities	yes
Common Behaviors	yes
Interactions	no
State Machines	no
Use Cases	no

Table 3.1: UML packages included in the foundational UML subset

As can be seen in Table 3.1, fUML supports the class diagram and the activity diagram. We want to go into more detail of the activities and actions package.

Activities

Table 3.2 shows the sub-packages included in the activities packages of UML and fUML and Figure 3.2 compares the dependencies between these sub-packages.

The basic compliance level L1 of fUML does not include any modeling concepts for activities. The Intermediate Activities sub-package is included on the intermediate layer L2. This package already merges the Basic Activities package as well as the Fundamental Activities package. On the complete level L3 the Extra Structured Activities package and the Complete Structured Activities package are included and therewith the Structured Activities package is merged into L3. The Complete Activities package is not separately supported in fUML.

Table 3.3 depicts which modeling concepts for activities are included in the foundational UML subset. We can see that only four concepts were excluded: Sequence Node, Flow Final Node, Central Buffer Node and Data Store Node.

Sub-package	*Compliance Level*		*Comment*
	UML	*fUML*	
Basic Activities	L1	(L2)	Required modeling concepts are merged into fUML package Intermediate Activities
Fundamental Activities	L1	(L2)	Required modeling concepts are merged into fUML package Intermediate Activities
Intermediate Activities	L2	L2	
Structured Activities	L2	(L3)	Required modeling concepts are merged into fUML packages Complete Structured Activities and Extra Structured Activities
Complete Activities	L3	-	Not separately supported in fUML
Extra Structured Activities	L3	L3	
Complete Structured Activities	L3	L3	

Table 3.2: Sub-packages of the activities language unit of UML and fUML included in the compliance levels L1 (basic), L2 (intermediate) and L3 (complete)

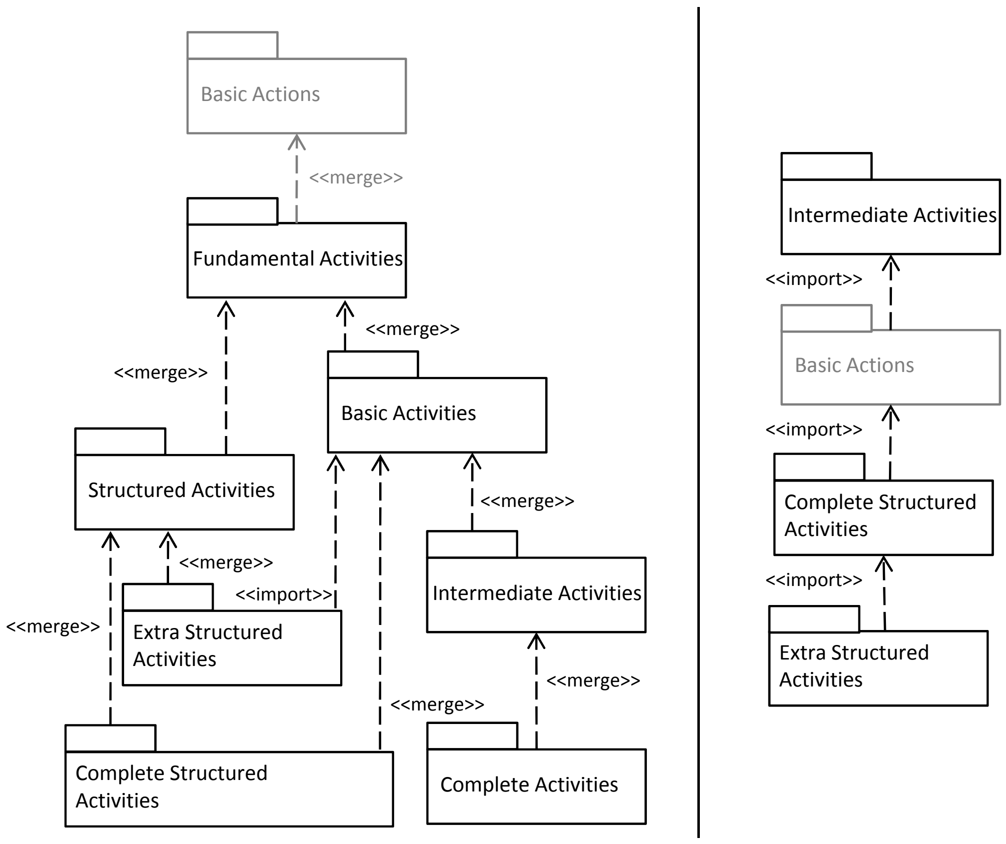


Figure 3.2: Comparison of the dependencies between the activities sub-packages of UML (left hand side) and fUML (right hand side) [19, 26]

UML	*Included in fUML?*
Activity	yes
Executable Nodes	
Structured Activity Node	yes
Conditional Node	yes
Loop Node	yes
Sequence Node	no
Expansion Region	yes
Control Nodes	
Initial Node	yes
Activity Final Node	yes
Decision Node	yes
Merge Node	yes
Fork Node	yes
Join Node	yes
Flow Final Node	no
Object Nodes	
Activity Parameter Node	yes
Expansion Node	yes
Central Buffer Node	no
Data Store Node	no
Activity Edges	
Control Flow	yes
Object Flow	yes

Table 3.3: Comparison of the modeling concepts of the activities language unit of UML and fUML

Actions

Table 3.4 shows the sub-packages included in the actions packages of UML and fUML. Figure 3.3 compares the dependencies between these sub-packages.

Like with the activities language unit, the basic compliance level L1 does not include any modeling concepts for actions. The intermediate level L2 includes the Basic Actions package as well as the Intermediate Actions package. The Structured Actions package is excluded from fUML due to the compactness criteria applied to the selection of the foundational UML subset. The complete level L3 includes the Complete Actions package.

In Table 3.5 the primitive actions supported by fUML are depicted. For this table the categorization into object-related, link-related, variable- and structural feature-related as well as communication-related actions of [10] was chosen. Particularly striking is the fact that the variable-related actions are all excluded. fUML therewith does not directly support variables.

Sub-package	*Compliance Level*		*Comment*
	UML	*fUML*	
Basic Actions	L1	L2	
Intermediate Actions	L2	L2	
Structured Actions	L2	-	Excluded from fUML
Complete Actions	L3	L3	

Table 3.4: Sub-packages of the actions language unit of UML and fUML included in the compliance levels L1 (basic), L2 (intermediate) and L3 (complete)

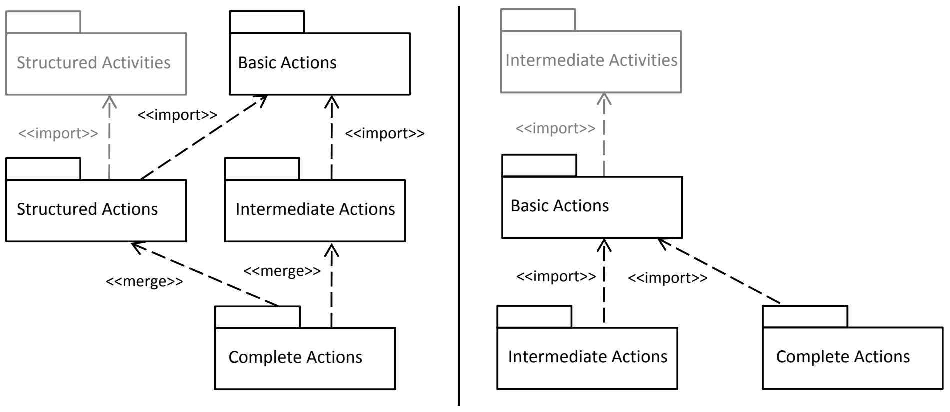


Figure 3.3: Comparison of the dependencies between the actions sub-packages of UML (left hand side) and fUML (right hand side) [19, 26]

UML	*Included in fUML?*
Object-related actions	
Create Object Action	yes
Destroy Object Action	yes
Read Self Action	yes
Test Identity Action	yes
Reclassify Object Action	yes
Read Is Classified Object Action	yes
Read Extent Action	yes
Start Classifier Behavior Action	yes
Start Object Behavior Action	yes
Link-related actions	
Create Link Action	yes
Create Link Object Action	no
Read Link Action	yes
Read Link Object End Action	no
Read Link Object End Qualifier Action	no
Clear Association Action	yes
Destroy Link Action	yes
Variable- and structural feature-related actions	
Add Variable Value Action	no
Read Variable Action	no
Clear Variable Action	no
Remove Variable Value Action	no
Add Structural Feature Value Action	yes
Read Structural Feature Action	yes
Clear Structural Feature Action	yes
Remove Structural Feature Value Action	yes
Value Specification Action	yes
Communication-related actions	
Accept Call Action	no
Accept Event Action	yes
Call Behavior Action	yes
Call Operation Action	yes
Broadcast Signal Action	no
Send Signal Action	yes
Send Object Action	no
Reply Action	no

UML	*Included in fUML?*
Other actions	
Opaque Action	no
Raise Exception Action	no
Reduce Action	yes
Unmarshall Action	no

Table 3.5: Comparison of the modeling concepts of the actions language unit of UML and fUML

3.3 Semantics of fUML - The Execution Model of fUML

The *execution model* is a formal, operational specification of the execution semantics of the foundational UML subset, defined in Java. It defines the execution semantics of all fUML modeling concepts as well as the fUML execution engine and environment. The execution model is itself an executable fUML model, namely a model of a fUML execution engine, and therewith it could be defined using fUML activity diagrams. But because this form of specification would lead to huge and hard-to-read diagrams, Java code is used to define the execution model instead of fUML activity diagrams. But the Java code can be seen as just another textual representation of the corresponding fUML activity diagram [20].

The structure of the execution model is the same as the structure of the abstract syntax, i.e., it includes the same packages and sub-packages and defines the semantics of the modeling concepts in the corresponding sub-package of the abstract syntax. Additionally the execution model includes a package called *Loci* which defines the fUML execution engine and environment. Thus the execution model incorporates the following packages.

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

Execution Engine and Execution Environment

As stated before, the package Loci defines the execution engine and the execution environment for executing fUML models. It contains the classes `Locus`, `Executor` and `ExecutionFactory`. The Loci package is structured according to the three compliance levels L1 (basic), L2 (intermediate) and L3 (complete). The sub-packages are named LociL1, LociL2 and LociL3. Figure 3.4 depicts the dependencies of the sub-packages of the Loci package to the other fUML packages.

LociL1 contains the biggest part of the Loci package and *LociL2* and *LociL3* only contain specialized classes of `ExecutionFactory`. The content of LociL1 is depicted in Figure 3.5.

The class `Executor` constitutes the execution engine. It offers three operations.

- **Execute**. This operation can be used to execute a behavior synchronously, whereat input can be provided and output is returned.
- **Evaluate**. With this operation value specifications can be evaluated and the corresponding value is returned.
- **Start**. This function executes a behavior asynchronously and returns a reference to the instance of the execution behavior.

An executor is located at maximal one `Locus` that represents a computer (real or virtual) that executes a fUML model. At the locus an arbitrary number of *extensional values* (instances of the class `ExtensionalValue`) may exist. Extensional values are objects and links which are created during the execution of a fUML model. They are persisted at the locus where the execution takes place, i.e., at the locus that houses the executor. An extensional value can only be located at one locus and still exists after the execution of a behavior is finished, i.e., extensional values can exist at the start of the behavior execution.

At a locus also an `ExecutionFactory` exists. It is used to instantiate so-called visitor classes. The fUML execution models makes use of the visitor design pattern. Therewith it adds the behavior to the fUML modeling concepts without adapting the metamodel that defines them. For every modeling concept of fUML, i.e., every metaclass of the abstract syntax of fUML, that should have a behavior, a so-called visitor class exists that specifies that behavior. In fUML we have to distinguish between three types of visitor classes, whereat every visitor class is derived from the class `SemanticVisitor`.

- **Execution**. This type of visitor class is used to add behavior to subclasses of the syntactic class `Behavior`. For instance the execution visitor class `ActivityExecution` defines how an activity (instance of class `Activity`) is executed.
- **Activation**. An activation visitor class specifies the semantics of an activity node. An example is the activation visitor class `CreateObjectActionActivation` that defines how a create object action (instance of class `CreateObjectAction`) behaves.
- **Evaluation**. Evaluation visitor classes are used to specify how value specifications are evaluated. For example the evaluation visitor class `LiteralBooleanEvaluation` is used to define how a boolean value (instance of class `LiteralBoolean`) is evaluated.

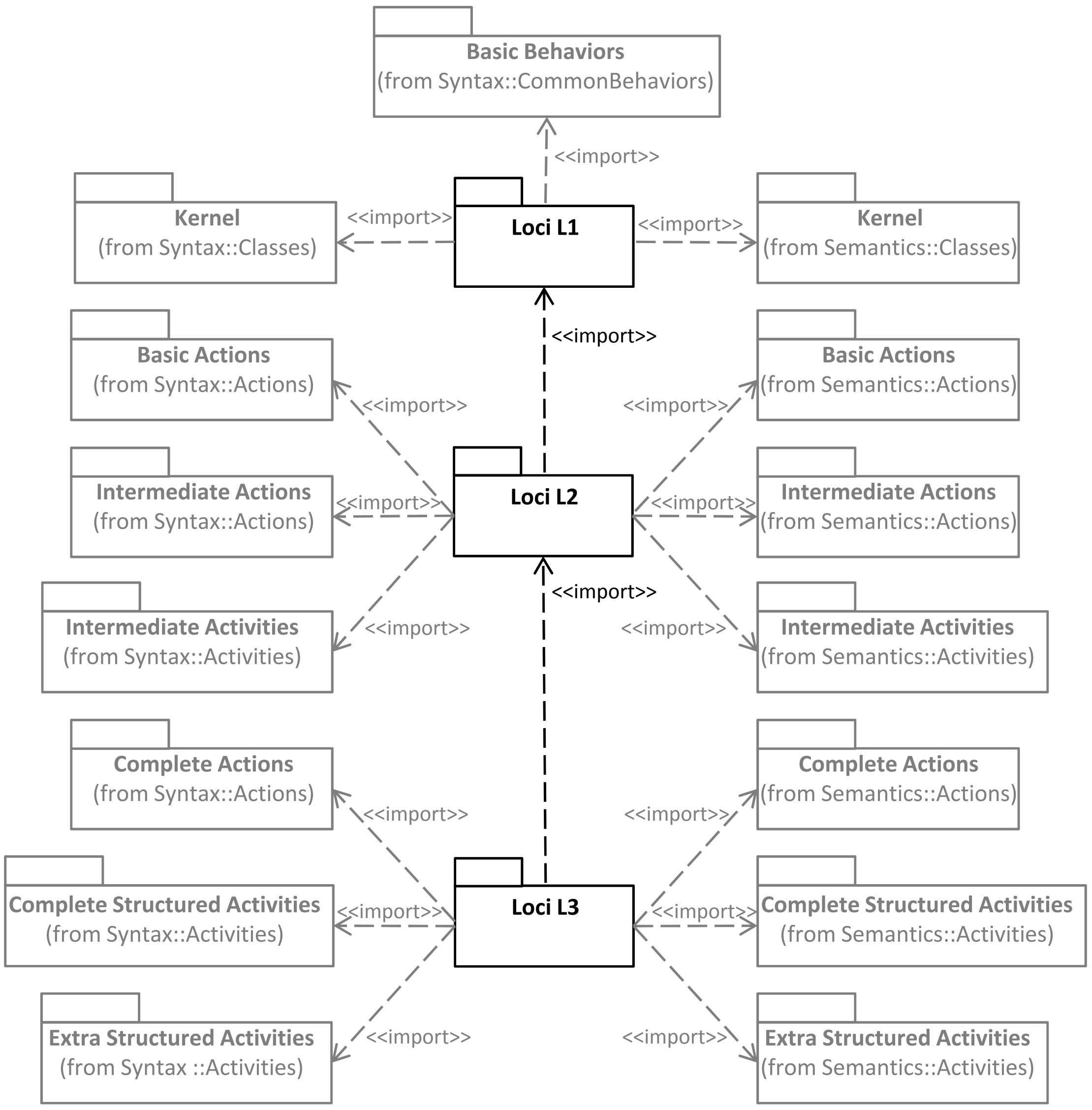


Figure 3.4: Dependencies of the Loci packages to other fUML packages [20]

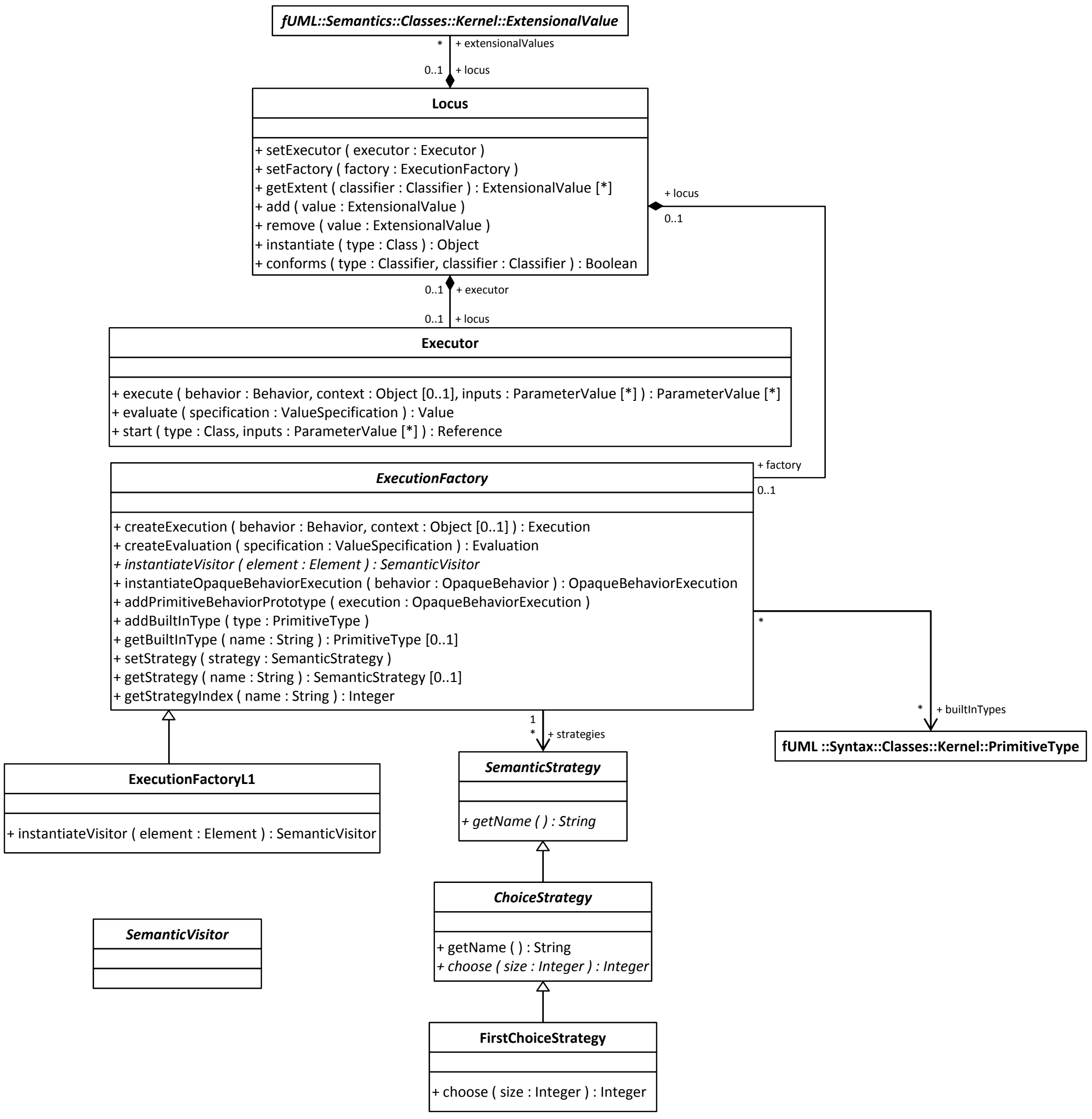

fUML::Semantics::Classes::Kernel::ExtensionalValue
* + extensionalValues
0..1 + locus
Locus
+ setExecutor (executor : Executor)
+ setFactory (factory : ExecutionFactory)
+ getExtent (classifier : Classifier) : ExtensionalValue [*]
+ add (value : ExtensionalValue)
+ remove (value : ExtensionalValue)
+ instantiate (type : Class) : Object
+ conforms (type : Classifier, classifier : Classifier) : Boolean
+ locus
0..1
0..1 + executor
0..1 + locus
Executor
+ execute (behavior : Behavior, context : Object [0..1], inputs : ParameterValue [*]) : ParameterValue [*]
+ evaluate (specification : ValueSpecification) : Value
+ start (type : Class, inputs : ParameterValue [*]) : Reference
ExecutionFactory
+ factory
0..1
+ createExecution (behavior : Behavior, context : Object [0..1]) : Execution
+ createEvaluation (specification : ValueSpecification) : Evaluation
+ instantiateVisitor (element : Element) : SemanticVisitor
+ instantiateOpaqueBehaviorExecution (behavior : OpaqueBehavior) : OpaqueBehaviorExecution
+ addPrimitiveBehaviorPrototype (execution : OpaqueBehaviorExecution)
+ addBuiltInType (type : PrimitiveType)
+ getBuiltInType (name : String) : PrimitiveType [0..1]
+ setStrategy (strategy : SemanticStrategy)
+ getStrategy (name : String) : SemanticStrategy [0..1]
+ getStrategyIndex (name : String) : Integer
*
* + builtInTypes
fUML ::Syntax::Classes::Kernel::PrimitiveType
1
* + strategies
ExecutionFactoryL1
+ instantiateVisitor (element : Element) : SemanticVisitor
SemanticStrategy
+ getName ( ) : String
ChoiceStrategy
+ getName ( ) : String
+ choose (size : Integer) : Integer
FirstChoiceStrategy
+ choose (size : Integer) : Integer
SemanticVisitor

Figure 3.5: Content of the LociL1 package [20]

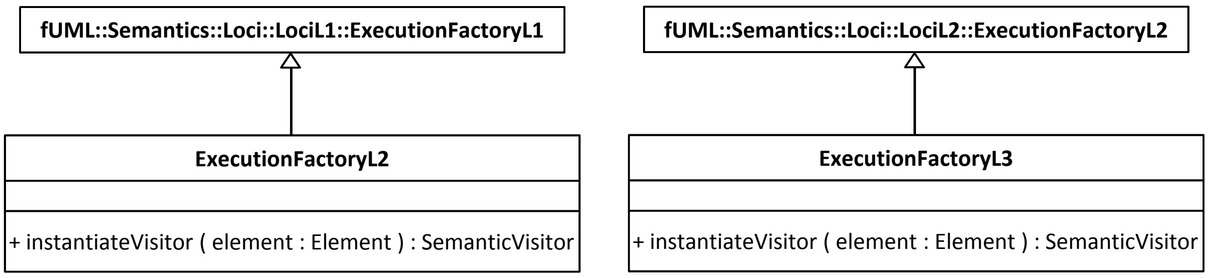

fUML::Semantics::Loci::LociL1::ExecutionFactoryL1
ExecutionFactoryL2
+ instantiateVisitor (element : Element) : SemanticVisitor
fUML::Semantics::Loci::LociL2::ExecutionFactoryL2
ExecutionFactoryL3
+ instantiateVisitor (element : Element) : SemanticVisitor

Figure 3.6: Content of the packages LociL2 and LociL3 [20]

As stated before each Loci sub-package has its own specialized execution factory to instantiate the visitor classes for the modeling concepts supported by the corresponding compliance level. The LociL2 and LociL3 packages only consist of these specialized execution factory classes (see Figure 3.6).

The execution factory maintains a set of primitive types (i.e., instances of the class `PrimitiveType`) as built in types for evaluating literal value specifications like `LiteralBoolean`.

For dealing with semantic variation points the fUML execution model uses the strategy design pattern. All strategy classes in fUML are sub-classes of `SemanticStrategy`. To find out with which variation point a semantic strategy is concerned the method `getName` is provided. A strategy class has to be registered at the execution factory using the operation `setStrategy`. The version 1.0 of fUML only considers two semantic variation points namely the dispatching of events and the dispatching of operations. Strategies that are concerned with the dispatching of events have to be subclasses of `GetNextEventStrategy` (contained in the package Semantics::CommonBehaviors::Communications) and strategies concerned with the dispatching of operations have to be subclasses of `DispatchStrategy` (located in the package Semantics::Classes::Kernel).

To specify nondeterministic behavior the strategy pattern is again used. An example for a nondeterministic behavior is a conditional node where more than one clause evaluates to true because only one clause body can be executed. To decide which of the possible execution paths in case of a nondeterministic behavior should be carried out one instance of a subclass of `ChoiceStrategy` can be registered at the execution factory that makes the decision each time nondeterministic behavior is identified. The LociL1 package provides a default strategy `FirstChoiceStrategy`.

To execute a fUML model the classes of the Loci package have to be instantiated, initialized and linked. The following components build-up the execution environment for executing fUML models [20]:

1. One `Locus` instance
2. One `Executor` instance that is linked to the locus using the method `setExecutor` of the locus
3. One `ExecutionFactory` instance on the appropriate conformance level that is linked to the locus using the method `setFactory` of the locus
4. For each primitive type Boolean, Integer, String and Unlimited Natural one instance of the class `PrimitiveType` has to be created and registered at the execution factory using the method `addBuiltInType`
5. One instance per strategy, i.e., instances of `ChoiceStrategy`, `DispatchStrategy` and `GetNextEventStrategy`, has to be created and register at the execution factory using the method `setStrategy`

Execution of an Activity

The execution engine carries out the following procedure to execute a fUML model.

1. **Provision of input on activity input parameter nodes**. Before the execution of an activity is started, the activity input values are provided to the activity execution.
2. **Identification of the enabled nodes**. In the first step the enabled nodes are identified by the execution engine. Enabled nodes are initial nodes, activity input parameter nodes and actions that have no incoming edges.
3. **Sending of control tokens to enabled nodes**. When the enabled nodes are identified, control tokens are sent to them.
4. **Execution of activity nodes**.
   a) **Check if activity node is ready to be executed**. When an activity node receives a token it is determined if this activity node is ready to be executed. This means that it is checked if all prerequisites for executing its behavior are fulfilled. More precisely it is checked if control tokens are available on all incoming control flow edges of that activity node and if the minimal necessary data tokens are available on the input pins.
   b) **Consumption of the tokens**. If an activity node is ready to be executed, i.e., the necessary control and data tokens are available, it consumes these offered tokens. This means that all tokens are removed from the incoming activity edges and are added to the activity node.

c) **Execution of the behavior of the activity node**. After the consumption of the tokens, the behavior of the activity node is executed. In case of an action this step may include that the output of that action is placed on its output pins in form of a data tokens.

d) **Sending of token to subsequent activity nodes**. When the behavior of an activity node has been executed, one control token per outgoing control flow edge is sent to the target of that control flow edge and in case of an action that has an output, the data tokens for that output are sent through the outgoing data flow edges of the corresponding output pins.

e) **Check if activity node should fire again**. After the execution of an activity node's behavior and the sending of tokens it is checked if the executed activity node can be executed again. To do so the steps a to d are repeated.

f) **Execution of subsequent activity nodes**. Because after executing an activity node, tokens are sent to subsequent activity nodes, these subsequent activity nodes may be ready to be executed too. So the steps a to e are performed for the activity nodes that received tokens.

5. **Provision of output on activity output parameter nodes**. When no activity node can be executed anymore, the execution of the activity is finished and the values on the activity output parameter nodes are provided.

An important point is that the execution semantics of fUML allows that activity nodes are executed concurrently. This means that tokens can be sent concurrently through outgoing control flow and data flow edges and that further the check if an activity node is ready to be executed, the consumption of provided tokens and the execution of its behavior can be seen as one thread. But the fUML execution model does not specify how to implement parallelism and any sequentially ordered, partial parallel or totally parallel execution is valid. This thesis considers the execution as being sequential and does not incorporate a threading model.

The presented procedure used by the execution engine of fUML to execute an activity is visualized in Figure 3.7 by means of an example. This exemplary activity diagram was already used in Chapter 2.5 as an example for ambiguities in the UML semantics (see Figure 2.6), so this is the solution to the question how often action C is executed. What should be noted is that like stated before it is assumed that the execution engine executes activities sequential, i.e., tokens are sent sequentially, actions are carried out sequentially etc. Thus the exemplary activity diagram is executed in the following steps:

1. The activity diagram consists of three actions called A, B and C and one join node. There are five control flow edges leading from action A to action C, from action B to action C, from action A to the join node, from action B to the join node and from the join node to action C.

2. First of all the enabled activity nodes are indentified. In this example action A and action B are enabled because they do not have any incoming edges. Because it is considered that

activity nodes are executed in a sequential order, a control token is sent to the first enabled action. So assuming that action A is the first enabled action, a control token is sent to action A.

3. After the reception of the control node, it is checked if action A can be executed. Because a control node is present and no more prerequisites have to be fulfilled in this case, action A is executed.

4. After the execution, action A sends control tokens to the outgoing control flow edges. We assume that first an token is sent to the control flow edge leading to action C. Action C receives the token and it is checked if the preconditions for executing action C are fulfilled. Because only one of the three incoming control flow edges provides control tokens, the preconditions are not fulfilled and action C can't be executed.

5. Now action A sends a control token to the join node and it is determined if it can be executed. This is not the case because only one incoming edge provides a token.

6. Action A is done with sending tokens and it can't be executed again. Because of this a control token is sent to the second enabled activity node action B.

7. After the reception of the control node, it is tested if all pre-conditions for executing action B are satisfied. This is the case and so action B is executed.

8. After execution, action B sends control tokens through the outgoing control flow edges. We assume that first a control token is sent to action C. Action C still can't be executed because one control flow edge still doesn't provide a control token.

9. Now action B sends a control flow token through the second outgoing control flow edge to the join node.

10. The join node can now be executed because both incoming edges provide control tokens. It consumes the two tokens and joins them.

11. The join node sends the joined control token to action C.

12. Because the pre-condition for executing action C is now fulfilled, i.e., all incoming control flow edges provide a control token, action C is executed. Because action C has no outgoing control or data flow edges, and action C, the join node and action B can't be executed again, the activity execution is finished.

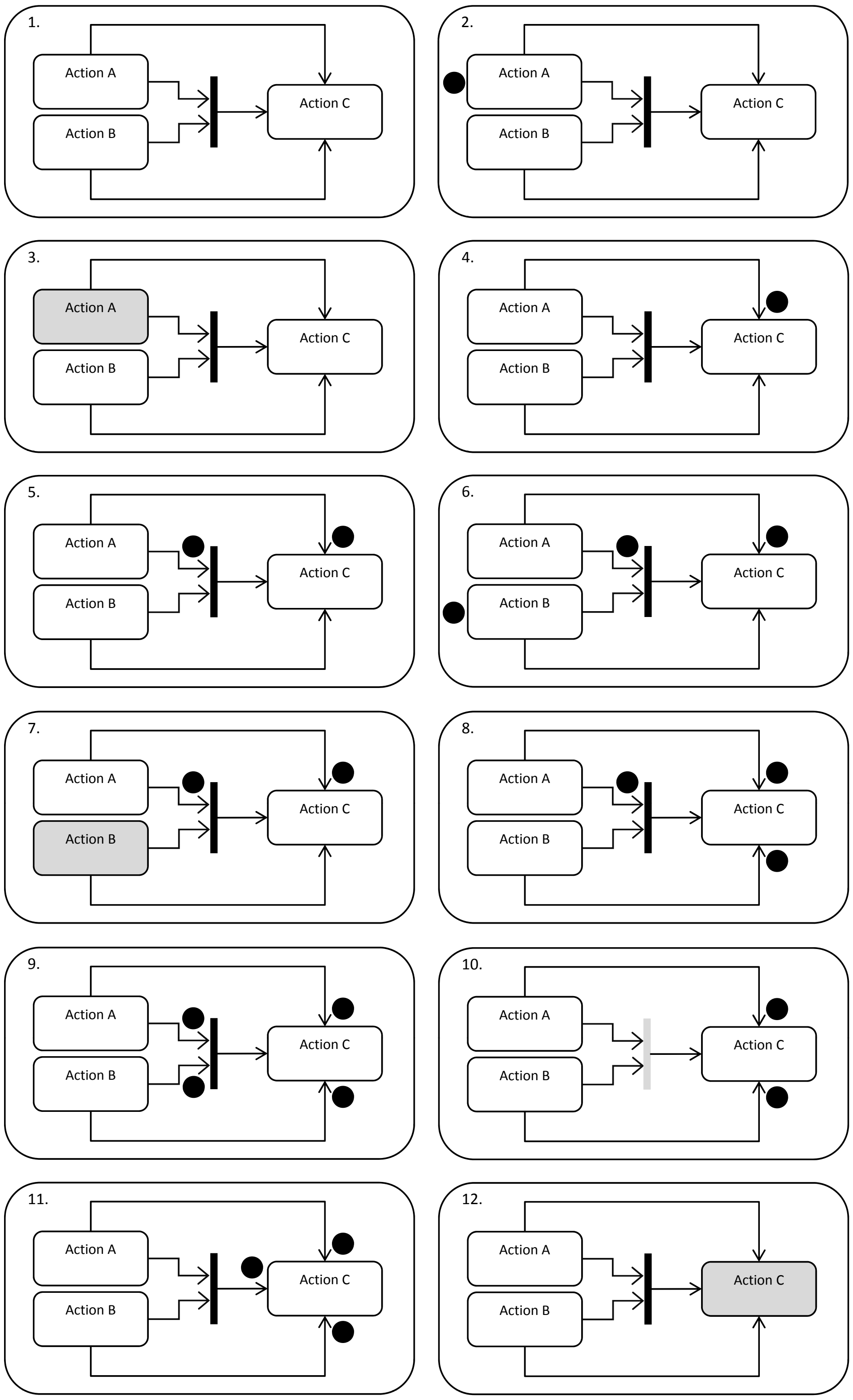


Figure 3.7: Example for the execution of a fUML model

The execution semantics of fUML is very similar to the Petri Net semantics [25]. UML activity diagrams can even be translated into petri nets and many transformation algorithms already exist. For example [11] specifies rules for mapping UML activity diagrams into colored petri nets. A detailed description of this transformation algorithm is out of scope of this thesis but three rules are picked out to illustrate that the execution semantics for activity diagrams specified by the fUML standard conforms to the petri net semantics:

1. Actions are mapped into transitions.
2. Input and output pins are mapped into places.
3. Control flow edges and data flow edges are mapped into arcs.
    a) If now two transitions are directly connected by an arc because in the activity diagram the corresponding two actions are directly connected, this arc is replaced by one place with one incoming and one outgoing arc.
    b) If now two places are directly connected by an arc because in the activity diagram the corresponding output pin and input pin are directly connected, this arc is replaced by a transition with one incoming and one outgoing arc.

Figure 3.8 depicts two examples for transforming activity diagrams into colored petri nets according to the presented rules.

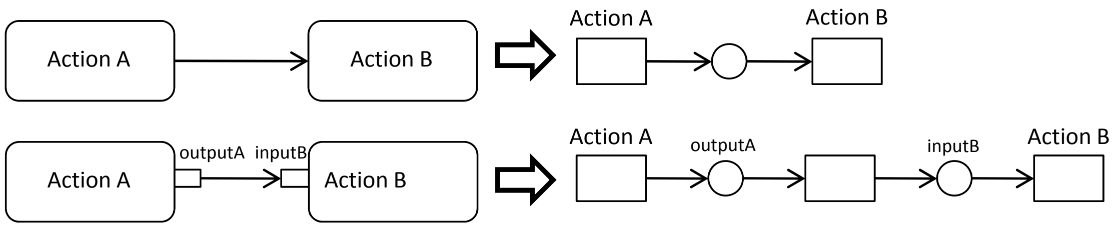


Figure 3.8: Examples for the transformation of activity diagrams into colored petri nets

CHAPTER 4

Eclipse Plug-In for Executing and Debugging fUML Models

4.1 Executing fUML Models

Having the possibility to execute the model of a system has significant advantages. By executing a model its behavior can be explored and therewith the understanding of the system can be improved. The execution of a model also serves as a means for testing. To test a model by executing it, the actual output of the execution can be compared with the expected output of the execution. If it is taken into consideration that in virtually every software project the system that should be implemented is modeled in the design phase and that these models are then used as basis for the implementation of the system, it can be concluded that finding and correcting errors in these models is essential to avoid their incorporation into the software product. The later an error in a system is detected, the more expensive is its correction. So errors should be corrected in an early stage and testing a model can be used to achieve this. Therewith the execution of models is a powerful tool. If we think a step ahead that MDD makes the software development process more model-centric and that code is generated from models and is used in the final software product, it is even more important to have a tool to test models.

Because the Unified Modeling Language is the de facto standard for modeling software systems, the execution of UML models leads to a "unified" tool for testing models of a software system. This thesis deals with the execution of UML activity diagrams because one fundamental principle of UML's semantics is that every behavior in a system is eventually caused by actions, i.e., every kind of behavior in UML is expressible as a sequence of actions. The execution semantics defined in the fUML standard was used to implement a prototypical model interpreter for UML activity diagrams or more precisely for fUML activity diagrams. This prototype of a model interpreter was implemented as Eclipse plug-in based on the Eclipse Modeling Framework. The architecture of this plug-in as well as the extensions and restrictions of the fUML metamodel and execution model are described in Chapter 4.3. The model interpreter deals with the execution of fUML activity diagrams that model the manipulation of objects and links, i.e., the creation and destruction of objects and links between these objects, the assignment of attribute values etc. The supported modeling concepts are also defined and described in detail in Chapter 4.3.

Procedure to Execute fUML Models

The procedure supported by the implemented model interpreter prototype to execute fUML activity diagrams consists of six steps whereat step 3, step 5 and step 6 are optional. This procedure is also depicted in Figure 4.1 in simplified form.

1. **Definition of the class diagram**. The first step consists of the definition of the class diagram. As mentioned before the model interpreter supports the execution of activity diagrams that model the manipulation of objects and links. This means that the executed activity diagram describes how objects of classes that are defined in a class diagram are created, destroyed, how links between objects are created and destroyed, how attribute values of objects are set and unset etc. So at first the class diagram has to be defined.

2. **Definition of the activity diagram**. Based on the defined class diagram the activity diagram is defined in the second step. This activity diagram describes the manipulation of objects and links using the supported primitive fUML actions like create object action, destroy object action, create link action, etc.

3. **Definition of the expected output**. In the third step the expected output of the execution of the activity diagram is defined. This expected output can later be compared with the actual output if desired. This step is optional. The model created in this step describes which objects exist, what values are assigned to their attributes and which links exist between the objects. But the diagram defined in this step is not an object diagram as defined by UML. It is a semantic model consisting of concepts that are used by the execution engine to define the runtime instances.

4. **Execution of the activity diagram**. The execution of the activity constitutes step 4. In this step the execution engine executes the behavior defined in the activity diagram.

5. **Storage of the activity output**. If the activity produces any output this output can be saved in step 5. Because not every activity produces an output also this step is optional.

6. **Comparison of the actual and expected output**. As the last step the possibly saved output of the executed activity can be optionally compared with the expected output that was defined in step 3 or it can be manually reviewed. If the actual output does not conform to the expected output, i.e., a deviation is discovered, the source of this deviation has to be determined. Three sources are thinkable. The first source is the class diagram, where for instance an association could be missing. The executed activity diagram is the second possible source. As an example control flow edges for regulating the control flow could be missing. Finally the expected output is the third thinkable source of a deviation because the behavior modeled in the activity diagram could be not completely thought through or

understood. Of course also a combination of these three sources is possible. If the source of the deviation between actual and expected output is identified it has to be corrected. This can include the correction of each of the three possible sources, i.e., the correction of the class diagram, the activity diagram and the diagram of the expected output, and the procedure starts anew.

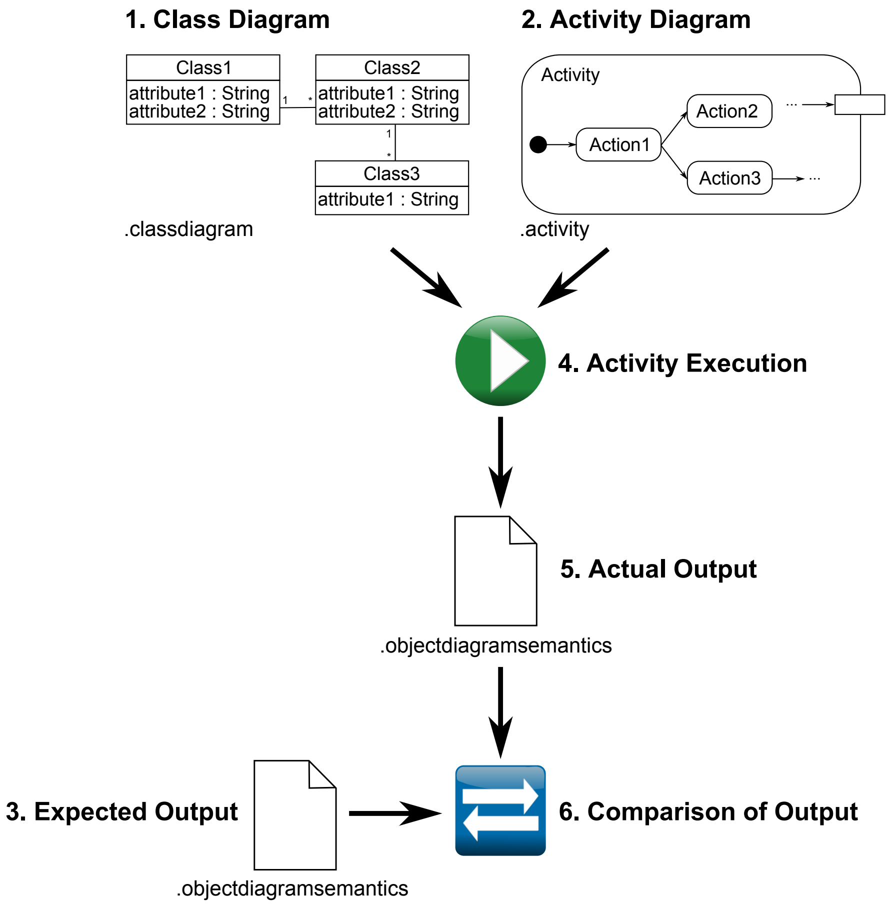
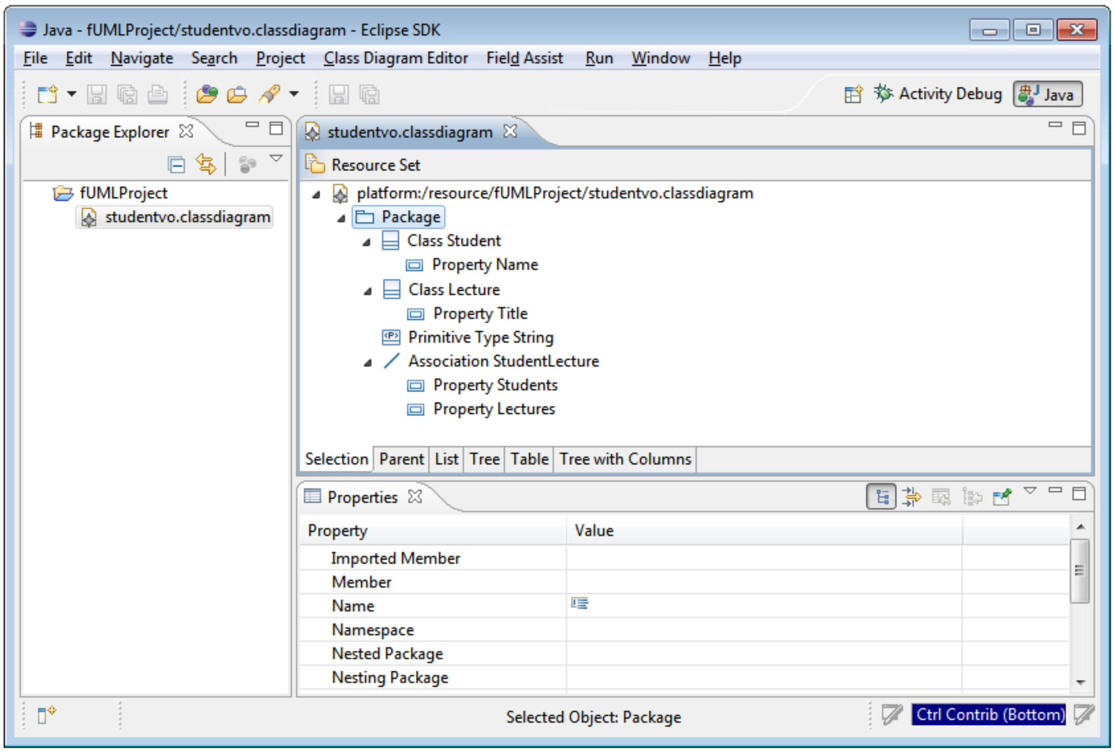


Figure 4.1: Procedure for executing activity diagrams implemented by the model interpreter prototype

Implementation of the Procedure as Eclipse Plug-In

The introduced procedure to execute fUML activity diagrams was implemented as an Eclipse plug-in that is based on EMF. Every step of the procedure is revisited to present the functionality and user interface of this plug-in. An example is used for a more detailed illustration.

1. **Definition of the class diagram**. For defining the class diagram the plug-in provides an editor. Class diagrams have the extension .classdiagram. First a wizard can be used to create a new instance of a class diagram. The model object, i.e., the root element of the model, is always a package. Then a tree-based editor can be used to define primitive types, classes and their properties as well as associations. These are the only model concepts for class diagrams that are supported by the interpreter. Figure 4.2 shows an example of a class diagram. The exemplary class diagram consists of a primitive type called `String` and two classes called `Student` and `Lecture`. An association `StudentLecture` exists between these two classes. The class `Student` has a property called `Name` of the type `String` and the class `Lecture` owns an attribute called `Title` of the type `String`. This class diagram is also depicted in Figure 4.3 in concrete syntax.


Java - fUMLProject/studentvo.classdiagram - Eclipse SDK
File Edit Navigate Search Project Class Diagram Editor Field Assist Run Window Help
Activity Debug
Java
Package Explorer
fUMLProject
studentvo.classdiagram
Selection Parent List Tree Table Tree with Columns
Properties
Selected Object: Package
Ctrl Contrib (Bottom)

Resource Set
- platform:/resource/fUMLProject/studentvo.classdiagram
  - Package
    - Class Student
      - Property Name
    - Class Lecture
      - Property Title
    - Primitive Type String
    - Association StudentLecture
      - Property Students
      - Property Lectures

Property	Value
Imported Member	
Member	
Name	
Namespace	
Nested Package	
Nesting Package	

Figure 4.2: Editor for defining class diagrams

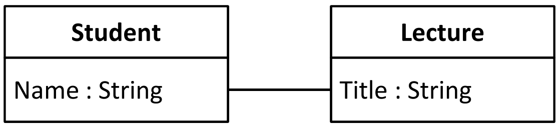
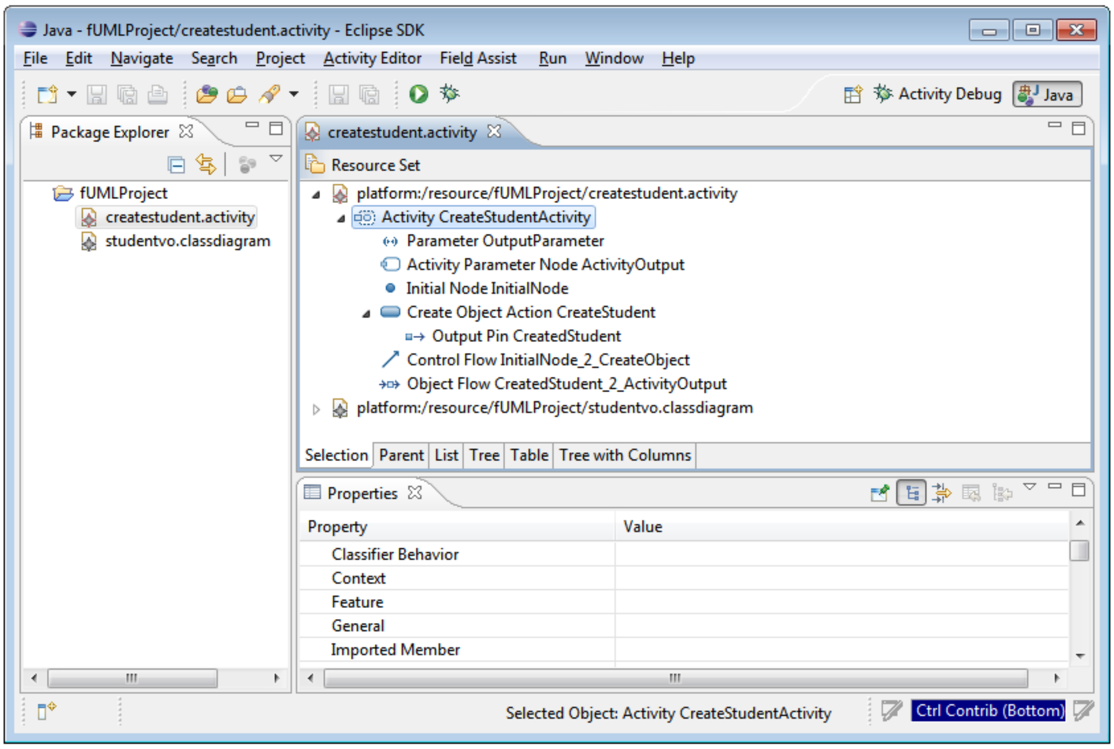


Figure 4.3: Concrete syntax of exemplary class diagram for illustrationg the editor for class diagrams

2. **Definition of the activity diagram**. To create an instance of an activity diagram again a wizard is provided that enables the instantiation of an activity diagram. The model object is an activity. A tree-based editor can then be used to add actions, control nodes, object nodes and edges to the activity. Not all kinds of actions and control nodes are supported by the plug-in. A detailed description of the supported modeling concepts is provided in Chapter 4.3. Activity diagrams have the extension .activity. Figure 4.4 shows an exemplary activity diagram. This activity starts with an initial node, then the control flow leads to a create object action where an instance of the class `Student` is created. This created instance is then provided as an output parameter. The concrete syntax of this activity diagram is depicted in Figure 4.5.


Java - fUMLProject/createstudent.activity - Eclipse SDK
File Edit Navigate Search Project Activity Editor Field Assist Run Window Help
Activity Debug
Java
Package Explorer
createstudent.activity
Selection Parent List Tree Table Tree with Columns
Properties
Ctrl Contrib (Bottom)

fUMLProject
createstudent.activity
studentvo.classdiagram

Resource Set
platform:/resource/fUMLProject/createstudent.activity
Activity CreateStudentActivity
Parameter OutputParameter
Activity Parameter Node ActivityOutput
Initial Node InitialNode
Create Object Action CreateStudent
Output Pin CreatedStudent
Control Flow InitialNode_2_CreateObject
Object Flow CreatedStudent_2_ActivityOutput
platform:/resource/fUMLProject/studentvo.classdiagram

Property	Value
Classifier Behavior	
Context	
Feature	
General	
Imported Member	

Selected Object: Activity CreateStudentActivity

Figure 4.4: Editor for defining activity diagrams

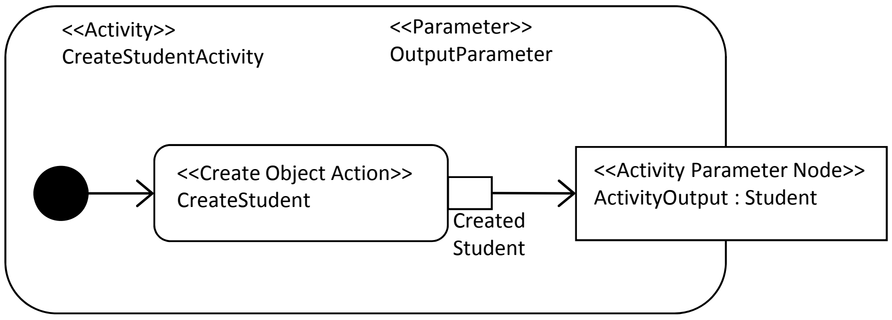
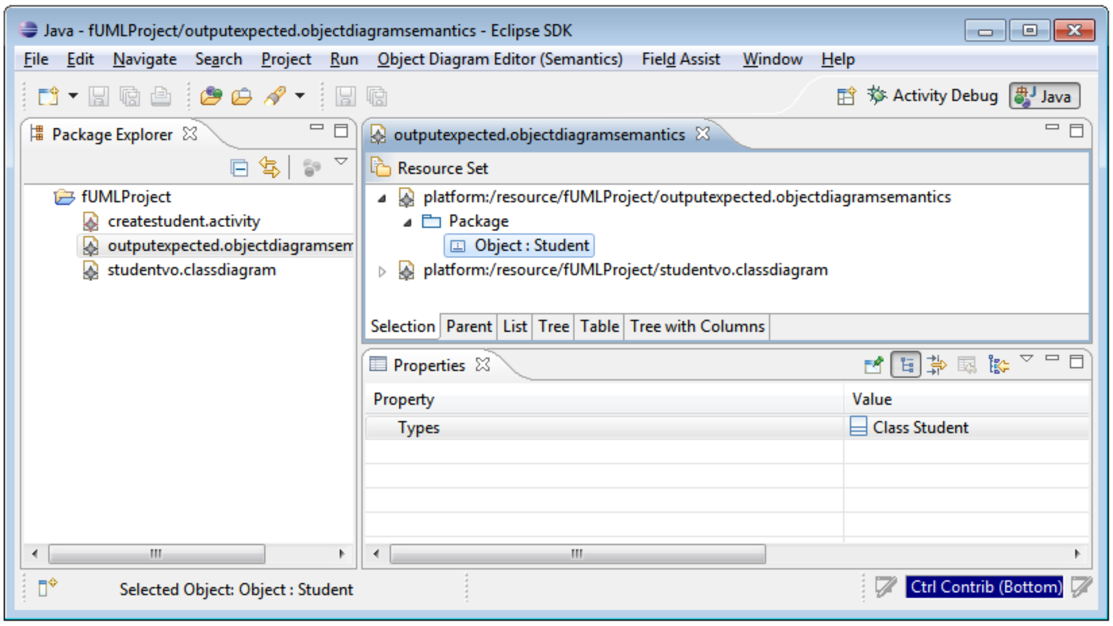


Figure 4.5: Concrete syntax of exemplary activity diagram for illustrating the editor for activity diagrams

3. **Definition of the expected output**. To define the expected output of an activity again a wizard is provided that can be used to instantiate a semantic object diagram. The model object is a semantic package and this package can include objects and links that can have feature values. The content of the semantic package can again be specified using a tree-based editor. Semantic object diagrams have the extension .objectdiagramsemantics. Figure 4.6 illustrates the expected output of the activity diagram depicted in Figure 4.4. Expected is an object of the type `Student`. Because this is a semantic diagram, it is not included in the UML standard but only in the fUML standard which does not define any concrete syntax, so there is no concrete syntax for a semantic object diagram.



fUMLProject
createstudent.activity
outputexpected.objectdiagramsem
studentvo.classdiagram

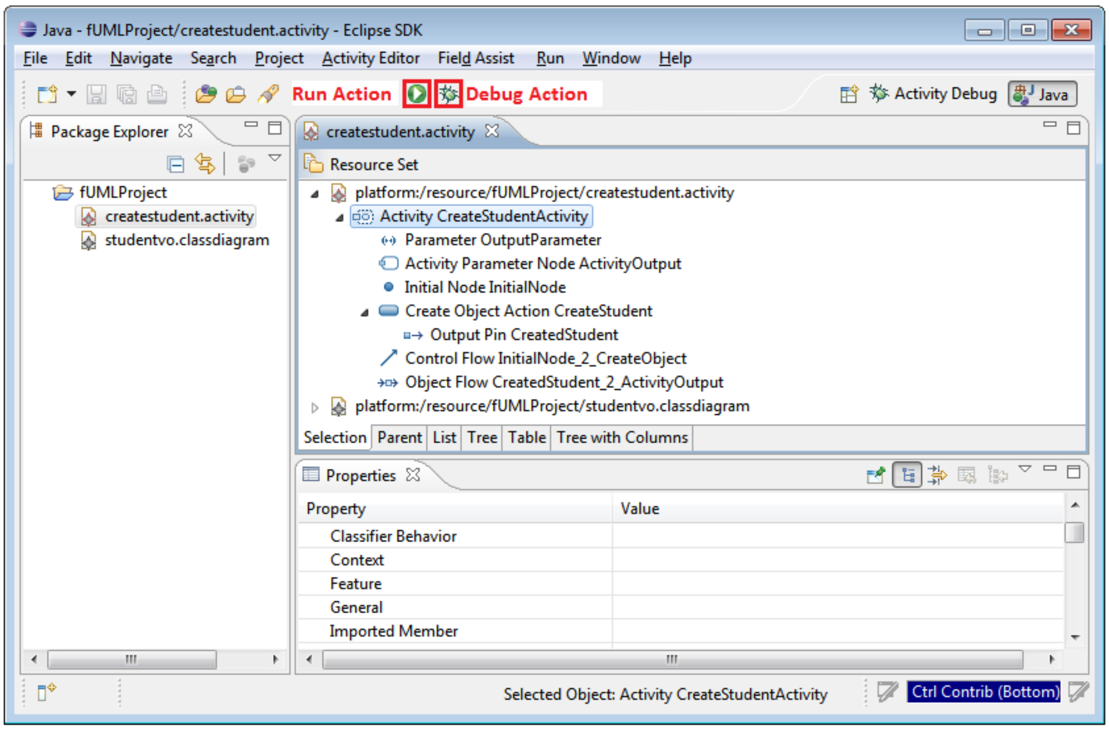
Resource Set
platform:/resource/fUMLProject/outputexpected.objectdiagramsemantics
Package
Object : Student
platform:/resource/fUMLProject/studentvo.classdiagram

Property	Value
Types	Class Student

Selected Object: Object : Student

Figure 4.6: Editor for defining semantic object diagrams

4. **Execution of the activity diagram**. For activity diagrams, i.e., files with the extension .activity, an action bar is available that provides the functionalities of executing and debugging activities. This action bar is visible if an activity diagram is opened. An example is displayed in Figure 4.7. The *Run* action can be used to execute an activity, whereas the *Debug* action can be used to debug an activity.


Java - fUMLProject/createstudent.activity - Eclipse SDK
File Edit Navigate Search Project Activity Editor Field Assist Run Window Help
Run Action
Debug Action
Activity Debug
Java
Package Explorer
createstudent.activity
Resource Set
Selection Parent List Tree Table Tree with Columns
Properties
Ctrl Contrib (Bottom)

fUMLProject
createstudent.activity
studentvo.classdiagram

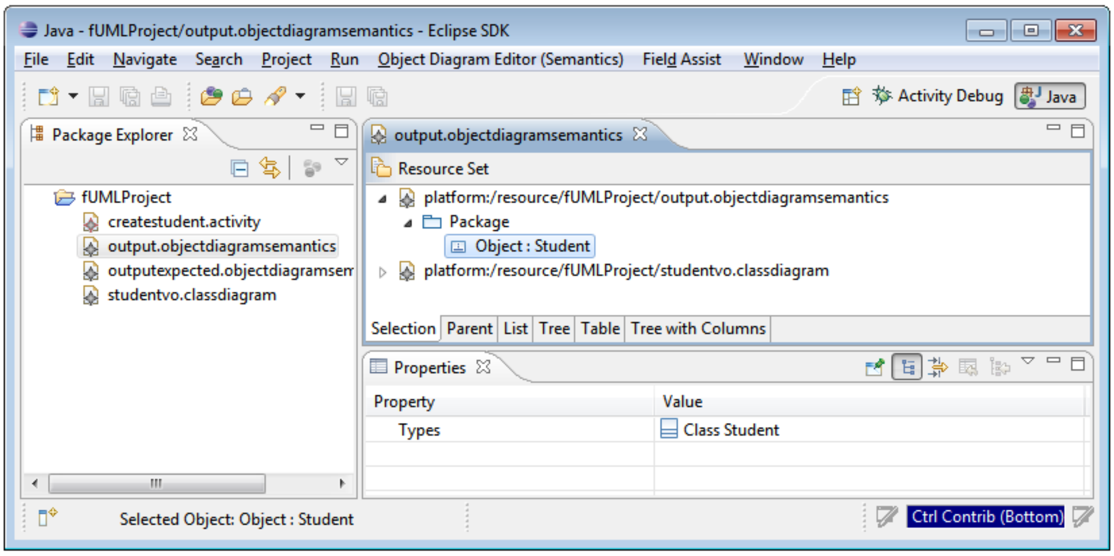
platform:/resource/fUMLProject/createstudent.activity
Activity CreateStudentActivity
Parameter OutputParameter
Activity Parameter Node ActivityOutput
Initial Node InitialNode
Create Object Action CreateStudent
Output Pin CreatedStudent
Control Flow InitialNode_2_CreateObject
Object Flow CreatedStudent_2_ActivityOutput
platform:/resource/fUMLProject/studentvo.classdiagram

Property	Value
Classifier Behavior	
Context	
Feature	
General	
Imported Member	

Selected Object: Activity CreateStudentActivity

Figure 4.7: Run action and debug action available for activity diagrams

5. **Storage of the activity output**. After executing the activity the user is asked if the created output - if any - should be saved and where. The saved activity output of the activity shown in Figure 4.4 is depicted in Figure 4.8. As expected an object of the class `Student` was created.


Java - fUMLProject/output.objectdiagramsemantics - Eclipse SDK
File Edit Navigate Search Project Run Object Diagram Editor (Semantics) Field Assist Window Help
Activity Debug
Java
Package Explorer
output.objectdiagramsemantics
Resource Set
Selection Parent List Tree Table Tree with Columns
Properties
Ctrl Contrib (Bottom)

fUMLProject
createstudent.activity
output.objectdiagramsemantics
outputexpected.objectdiagramsem
studentvo.classdiagram

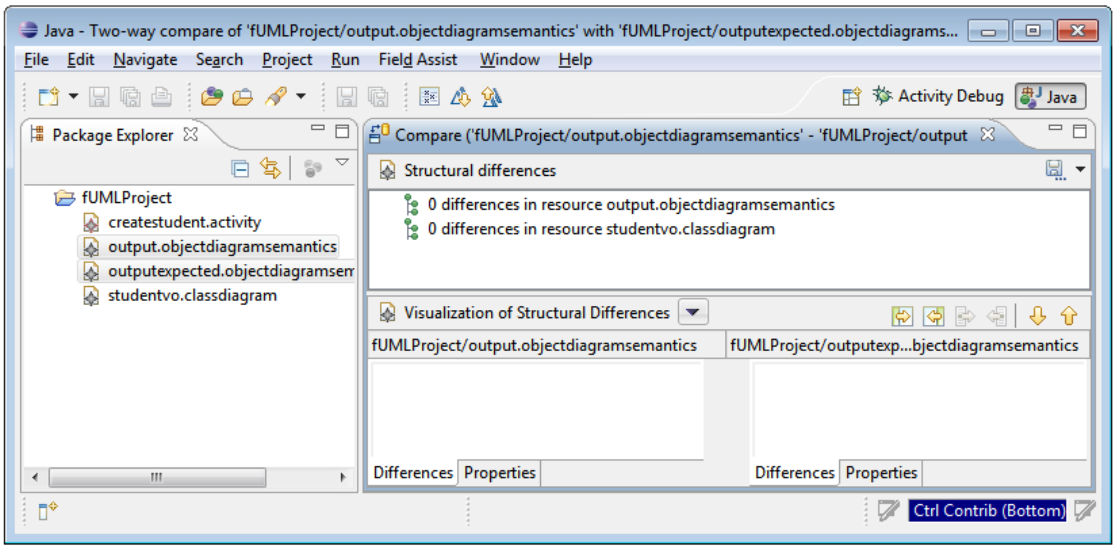
platform:/resource/fUMLProject/output.objectdiagramsemantics
Package
Object : Student
platform:/resource/fUMLProject/studentvo.classdiagram

Property	Value
Types	Class Student

Selected Object: Object : Student

Figure 4.8: Saved activity output

6. **Comparison of the actual and expected output**. If now the expected output depicted in Figure 4.6 of step 3 and the actual output depicted in Figure 4.8 of step 5 are compared, it can be seen that they are identical. Therewith the test of the activity diagram depicted in Figure 4.4 of step 2 was positive, i.e., the activity diagram is correct. The comparison can also be done automated using EMF Compare (see Figure 4.9).


Java - Two-way compare of 'fUMLProject/output.objectdiagramsemantics' with 'fUMLProject/outputexpected.objectdiagrams...
File Edit Navigate Search Project Run Field Assist Window Help
Activity Debug
Java
Package Explorer
Compare ('fUMLProject/output.objectdiagramsemantics' - 'fUMLProject/output
Structural differences
Visualization of Structural Differences
Differences Properties
Differences Properties
Ctrl Contrib (Bottom)

fUMLProject
createstudent.activity
output.objectdiagramsemantics
outputexpected.objectdiagramsem
studentvo.classdiagram

0 differences in resource output.objectdiagramsemantics
0 differences in resource studentvo.classdiagram

fUMLProject/output.objectdiagramsemantics
fUMLProject/outputexp...bjectdiagramsemantics

Figure 4.9: Comparison of expected and actual output of activity execution using EMF Compare

4.2 Debugging fUML Models

Chapter 4.1 pointed out that executing models is a means for better understanding models and their underlying systems and that the execution can serve as a tool for testing models by comparing the expected and the actual output of the execution. Debugging models can lead to an even better understanding and a more thorough testing of models. By debugging a model step-by-step the execution can be observed in detail.

Because of these advantages the implemented model interpreter prototype also provides the functionality of debugging fUML activity diagrams. This functionality was already mentioned in Chapter 4.1. The action bar attached to the editor for activity diagrams provides the debug action which starts the debugging of the opened activity diagram (see Figure 4.7). For the debugging an extra perspective called *activity debug perspective* is provided. Much like the debug perspective provided by Eclipse for the debugging of Java programs, also the activity debug perspective opens up automatically when the debug action for an activity diagram is executed. The activity debug perspective also provides similar functionality as the debug perspective for programs. It consists of four views:

1. **Activity Debug View**. The activity debug view is the main view of the activity debug perspective. It displays the activity diagram which is currently debugged and shows the current status of the debugging process by highlighting the next activity node to be executed. I.e., after the debugging is started the execution of the activity is initialized and paused before the first activity node is executed. This activity node is then highlighted. This view provides four actions which can then be used to control the further debugging process. The functionality of these control mechanisms also resembles those of the debugging functionality for programs:

    a) **Forward Action**. The execution of this action causes that the next activity node, i.e., the activity node that is highlighted in the activity debug view, is executed. After the execution of this activity node the execution of the activity is again paused and the next activity node which is to be executed next is highlighted in the activity debug view. Thus this action can be used to execute the activity diagram stepwise, i.e., activity node by activity node.

    b) **Resume Action**. The resume action causes that the activity is carried out until either the end of the activity is reached or an activity node is reached for which a breakpoint was set.

    c) **Breakpoints**. As indicated, the plug-in provides the possibility to set and unset breakpoints for activity nodes. Together with the resume action, breakpoints simplify the debugging of an activity. If the resume action is triggered - and breakpoints are set - the execution of the activity is resumed until an activity node is reached for which a breakpoint was set. This means that the execution pauses just before the execution of this activity node with the set breakpoint.

    d) **Stop Action**. This action stops the debugging of the activity, i.e., it cancels the execution of the activity.

2. **Runtime Objects View**. This view shows which objects are present during the debugging of the activity, i.e., it displays the runtime objects, the value set for their attributes and the existing links between these objects. This view is refreshed after each execution of an activity node so that the direct effect of every single activity node is instantly presented to the user. The runtime objects are displayed in the same way as they would be displayed by the editor for semantic object diagrams.

3. **Trace View**. The trace view shows which activity nodes were already executed and the chronology of their execution. Therewith each activity node appears as often in the trace view as it has been executed until then. The trace view is also refreshed after each execution of an activity node, i.e., each executed activity node is instantly added to the trace. The activity nodes are displayed in the trace view in the same way as they are in the activity editor.

4. **Activity Editor**. The activity debug perspective also includes the activity editor that displays the activity diagram that is debugged.

If the debugging of the activity finished, i.e., the activity was completely executed and not cancelled using the stop action, and an output was generated, this output can be saved.

To visualize the debugging functionality of the plug-in and its user interface in more detail, the exemplary activity diagram depicted in Figure 4.10 is used. Here an instance of the class `Student` is created and the `String` value `Tanja` is assigned to the attribute `Name`. Further two instances of the class `Lecture` are created. For one the `String` value `Object-Oriented Modeling` (OOM) is assigned to the `Title` attribute, for the other the `String` value `Model Engineering` (ME) is assigned to the `Title` attribute. Also two links are created, namely between the object of the type `Student` and the two objects of the type `Lecture`.

Figure 4.11 shows the activity debug perspective for the debugging of this activity diagram. In the upper area of this perspective the activity editor is located and in the lower region the three views are situated whereat the activity debug view is positioned on the left-hand side and the runtime objects view and the trace view are positioned beneath each other on the right-hand side. These four views are now presented in more detail with the help of the introduced example.

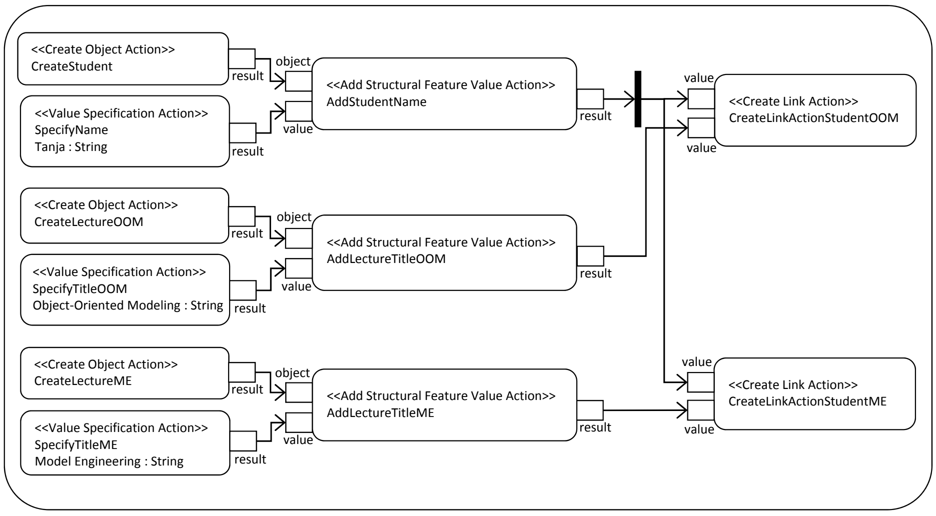
Activity Editor. The activity editor displays the debugged activity.

Activity Debug View. The activity debug view shows that the execution of the activity is paused at the create link action that creates the link between the object of the type `Student` and the object of the type `Lecture` which has the `String` value `Object-Oriented Modeling` assigned to its `Title` attribute. This is indicated by the green background of this action. This means that this create link action was not executed yet but will be executed next after the execution of the activity is resumed using the resume action or the forward action. This create link action is also an example for an activity node for which a breakpoint is set. That a breakpoint is set is denoted by a read point ("breakpoint") situated on the upper left side of the icon of the activity node.

Trace View. The trace view visualizes that seven activity nodes have been already executed in the displayed chronology:

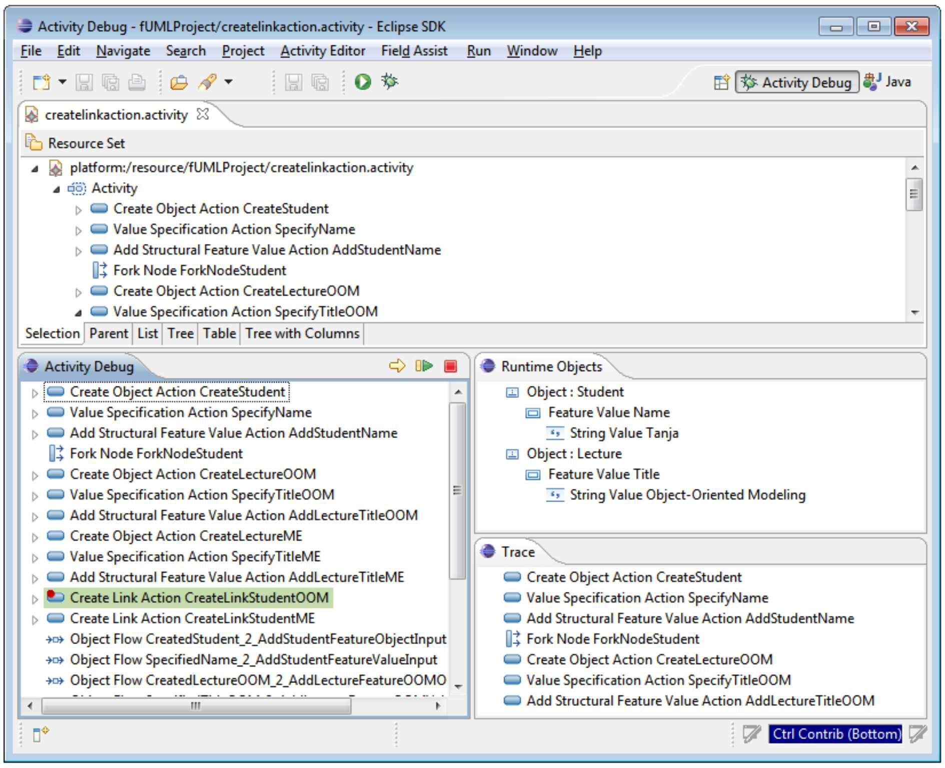
1. Instantiation of an object of the type `Student` (create object action "CreateStudent")
2. Specification of the `String` value `Tanja` (value specification action "SpecifyName")
3. Assignment of the specified `String` value `Tanja` to the attribute `Name` of the `Student` object (add structural feature value action "AddStudentName")
4. Provision of the `Student` object for the two create link actions (fork node "ForkNodeStudent")
5. Instantiation of an object of the type `Lecture` (create object action "CreateLectureOOM")
6. Specification of the `String` value `Object-Oriented Modeling` (value specification action "SpecifyTitleOOM")
7. Assignment of the specified `String` value `Object-Oriented Modeling` to the attribute `Title` of the `Lecture` object (add structural feature value action "AddLectureTitleOOM")

Runtime Objects View. Because of the already executed actions the runtime instances are the `Student` object with the `String` value `Tanja` assigned to the attribute `Name` and the `Lecture` object with the `String` value `Object-Oriented Modeling` assigned to the attribute `Title`. These runtime instances are displayed in the runtime objects view.



<<Create Object Action>> CreateStudent
result
object
<<Value Specification Action>> SpecifyName Tanja : String
result
value
<<Add Structural Feature Value Action>> AddStudentName
result
value
<<Create Link Action>> CreateLinkActionStudentOOM
value
<<Create Object Action>> CreateLectureOOM
result
object
<<Value Specification Action>> SpecifyTitleOOM Object-Oriented Modeling : String
result
value
<<Add Structural Feature Value Action>> AddLectureTitleOOM
result
<<Create Object Action>> CreateLectureME
result
object
<<Value Specification Action>> SpecifyTitleME Model Engineering : String
result
value
<<Add Structural Feature Value Action>> AddLectureTitleME
result
value
<<Create Link Action>> CreateLinkActionStudentME
value

Figure 4.10: Exemplary activity diagram to illustrate the debugging functionality of the model interpreter prototype



Activity Debug - fUMLProject/createlinkaction.activity - Eclipse SDK
File Edit Navigate Search Project Activity Editor Field Assist Run Window Help
Activity Debug
Java
createlinkaction.activity
Resource Set
Selection Parent List Tree Table Tree with Columns
Activity Debug
Runtime Objects
Trace
Ctrl Contrib (Bottom)

platform:/resource/fUMLProject/createlinkaction.activity
Activity
Create Object Action CreateStudent
Value Specification Action SpecifyName
Add Structural Feature Value Action AddStudentName
Fork Node ForkNodeStudent
Create Object Action CreateLectureOOM
Value Specification Action SpecifyTitleOOM

Activity Debug:
Create Object Action CreateStudent
Value Specification Action SpecifyName
Add Structural Feature Value Action AddStudentName
Fork Node ForkNodeStudent
Create Object Action CreateLectureOOM
Value Specification Action SpecifyTitleOOM
Add Structural Feature Value Action AddLectureTitleOOM
Create Object Action CreateLectureME
Value Specification Action SpecifyTitleME
Add Structural Feature Value Action AddLectureTitleME
Create Link Action CreateLinkStudentOOM
Create Link Action CreateLinkStudentME
Object Flow CreatedStudent_2_AddStudentFeatureObjectInput
Object Flow SpecifiedName_2_AddStudentFeatureValueInput
Object Flow CreatedLectureOOM_2_AddLectureFeatureOOMO

Runtime Objects:
Object : Student
Feature Value Name
String Value Tanja
Object : Lecture
Feature Value Title
String Value Object-Oriented Modeling

Trace:
Create Object Action CreateStudent
Value Specification Action SpecifyName
Add Structural Feature Value Action AddStudentName
Fork Node ForkNodeStudent
Create Object Action CreateLectureOOM
Value Specification Action SpecifyTitleOOM
Add Structural Feature Value Action AddLectureTitleOOM

Figure 4.11: Debug perspective of the model interpreter prototype

4.3 Architecture of the Plug-In

The functionality and the user interface of the implemented prototypical model interpreter were presented in Chapter 4.1 and Chapter 4.2. This chapter is concerned with the architecture of this plug-in and with the restrictions as well as extensions of the fUML metamodel and the fUML execution model.

Architecture

The plug-in was built using the Eclipse Java framework and code generation facility EMF. The implementation encompassed the following three steps:

1. **Ecore model**. OMG provides the fUML metamodel as CMOF (Complete Meta Object Facility) model, i.e., the fUML metamodel is defined using the CMOF meta-metamodel. With Eclipse this CMOF model was converted into an Ecore model.

2. **Model code**. The Ecore model of fUML was then used to generate the model code which contains the program logic of the model interpreter. The methods of the semantics classes were implemented like defined in the formal specification of fUML.

3. **Edit code and editor code**. The next step was then to generate the edit and editor code of the Ecore model of fUML in order to build the editors for creating fUML models, i.e., the editors for class diagrams, activity diagrams and semantic object diagrams. The edit code and the editor code were then adapted to restrict the supported modeling concepts and to adjust the user interface (for example icons were defined for the different modeling concepts).

The architecture of the plug-in is also depicted in Figure 4.12. The component *fUML.editor* constitutes the editors as well as the wizards provided to define fUML models. It uses the *fUML.edit* component that contains the item provider classes. This component uses the component called *fUML* which contains the model code.

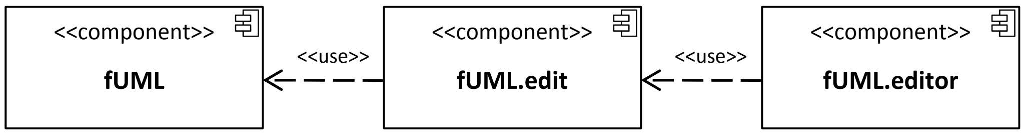


Figure 4.12: Architecture of the plug-in

Conformance Statement

The fUML standard suggests that the conformance of an execution tool to the fUML specification is summarized in a so-called *conformance statement*. The items that should be concluded in such a conformance statement are also specified by the standard.

This is the conformance statement of the implemented model interpreter prototype:

Conformance Level. All the modeling concepts supported by the model interpreter prototype are situated on the compliance level L1 or on the compliance level L2. Because of this an execution factory of the level L2, i.e., an instance of `ExecutionFactoryL2`, is used. But not all elements of these compliance levels are supported. The supported modeling concepts are described below. If a model that includes elements that are not supported is provided as input to the execution engine a runtime error occurs.

Model Library. The fUML standard contains a model library that defines primitive behaviors like functions for Integer values that can be used in a fUML model. The implementation of this model library is not mandatory. No elements of the fUML model library are supported by the implemented prototype.

Abstract Syntax Mapping. The models created by using the provided editors are saved in the XMI format which is the input format supported by the model interpreter prototype. The provided input model is loaded into an in-memory representation consisting of instances of the Java classes generated from the metamodel of fUML.

Semantic Value Mapping. The model interpreter prototype does not allow the provision of input values during the execution of an activity. All necessary values have to be provided within the model. Values are internally represented by instances of the corresponding Java classes generated from the metamodel of fUML.

Execution Environment Mapping. The locus concept is directly implemented as Java class generated from the metamodel of fUML:

- The execution of an activity takes place at a single locus, so the execution environment includes only one instance of the class `Locus`.
- The execution environment and therewith its locus is instantiated for each activity execution. Because of this created extensional values are not persisted.
- No objects are pre-instantiated at the locus.

Semantic Conformance. The `Executor` class is strictly implemented according to the fUML formal specification. However, the plug-in only supports the synchronous execution of an activity by using the `execute` method of the executor.

Semantic Constraints. The fUML standard does not restrict the semantics of time, the semantics of concurrency and the semantics of inter-object communication mechanisms. These semantic areas are constrained by the model interpreter prototype as follows:

- **Semantics of time**. The model interpreter prototype does not incorporate an explicit time model.
- **Semantics of concurrency**. The prototype does not incorporate a threading model. The actions of an activity are executed sequentially.
- **Semantics of inter-object communication mechanisms**. Ordinary Java operation calls within a single Java Virtual Machine are used by the model interpreter prototype as inter-object communications mechanism.

Semantic Variation. As described in Chapter 3.3, the fUML execution engine uses semantic strategy classes in order to deal with semantic variation points as well as with nondeterministic behavior. The implemented interpreter uses the default semantic strategy classes defined in the fUML standard.

Supported Modeling Concepts

Not all modeling concepts that are provided by fUML are also supported by the built model interpreter prototype. The interpreter was intended to execute and debug fUML activity diagrams that model the manipulation of objects and links, i.e., the creation and destruction of objects and links between these objects, the assignment of attribute values etc. Because of this only modeling concepts that are relevant for this kind of models are supported by the plug-in. The editor code was adjusted according to these restrictions so that the user can only add the selected modeling concepts to a diagram. But also extensions had to be made in order to implement the prototypical model interpreter. Following the supported modeling concepts of the different diagram types are presented.

Class Diagram

The Figures 4.13, 4.14 and 4.15 depict excerpts of the fUML metamodel concerning class diagrams that include the supported modeling concepts and their most important attributes.

The model element, i.e., the root element of a class diagram, is always an instance of the metaclass `Package`. As depicted in Figure 4.13 the following three modeling concepts can be added to that package: `Class`, `Association`, `PrimitiveType`. These “packageable” elements are subclasses of `NamedElement` and `Element` and therewith have an attribute called `name` and can possess any number of instances of the metaclass `Comment` that has an attribute called `body` to define a comment for elements.

Figure 4.14 shows the relationship between the modeling concepts `Class`, `Property` and `Association`. A class can have an arbitrary number of properties, i.e., instances of the metaclass `Property`. The metaclass `Property` is a subclass of the metaclasses `NamedElement` and `TypedElement`. Because of this a property owns an attribute called `name` and it can reference an instance of the metaclass `Type` that denotes the type of the property. Because also the metaclass `MultiplicityElement` is a superclass of `Property`, Integer literals or Unlimited Natural literals can be used to define the lower and upper bound of the multiplicity interval of a property as depicted in Figure 4.15.

Instances of the metaclass `Association` can be used to model associations between classes. This association object must own one property for each class participating in the association and references these so-called member ends as well as the classifiers that are used as types of these ends. In case of a binary association where both association ends are navigable, the two properties that represent the association ends reference each other.

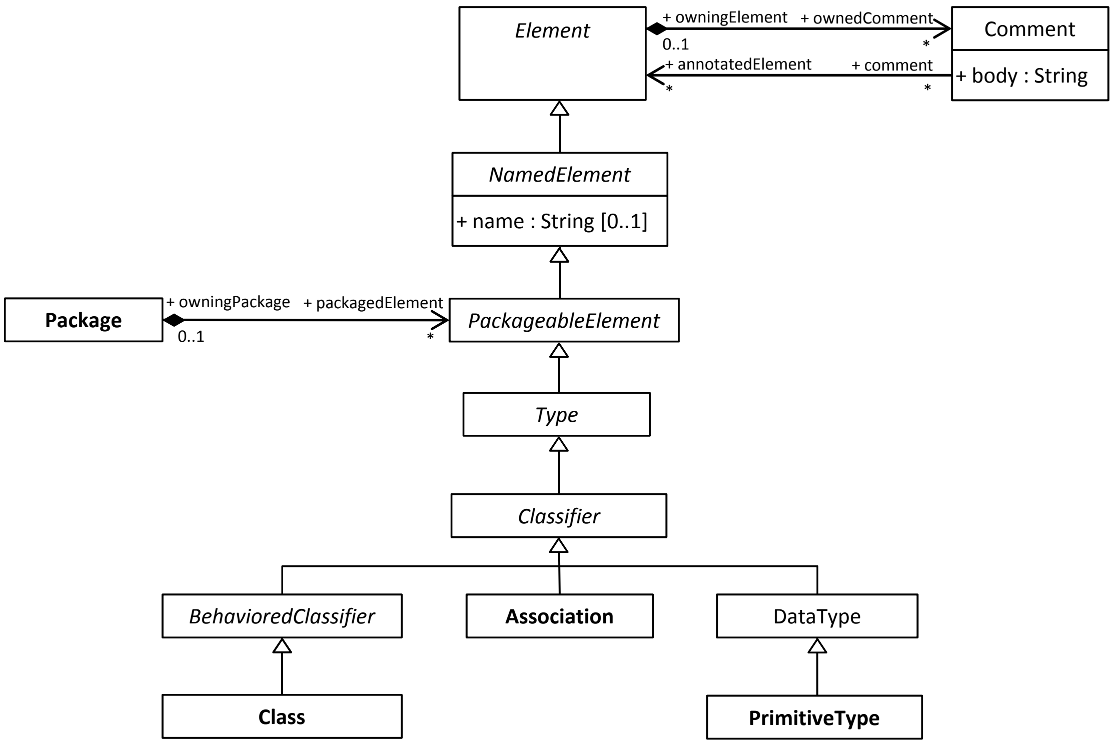


Figure 4.13: Class diagram elements supported by the plug-in - Packageable elements [20]

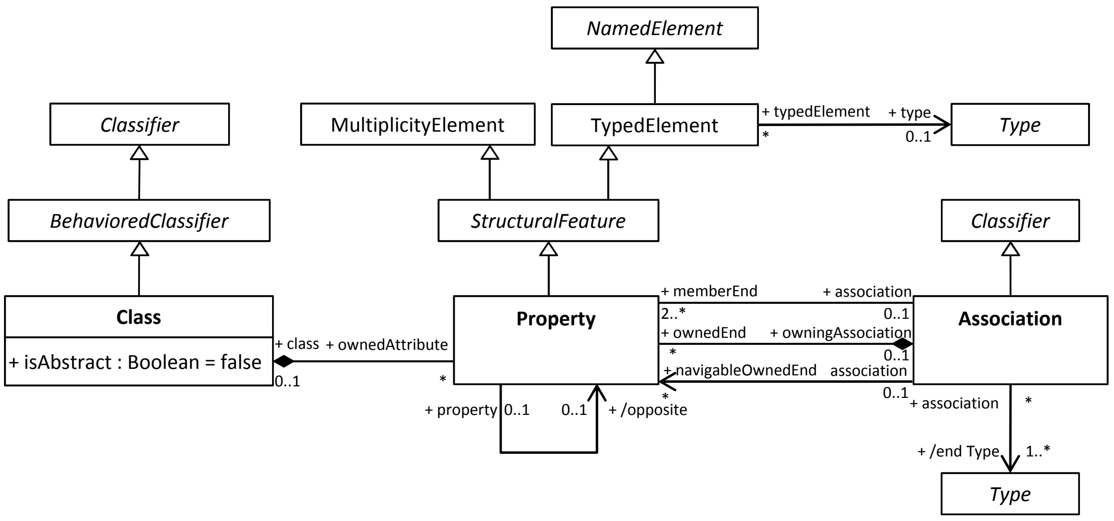


Figure 4.14: Class diagram elements supported by the plug-in - Classes, properties and associations [20]

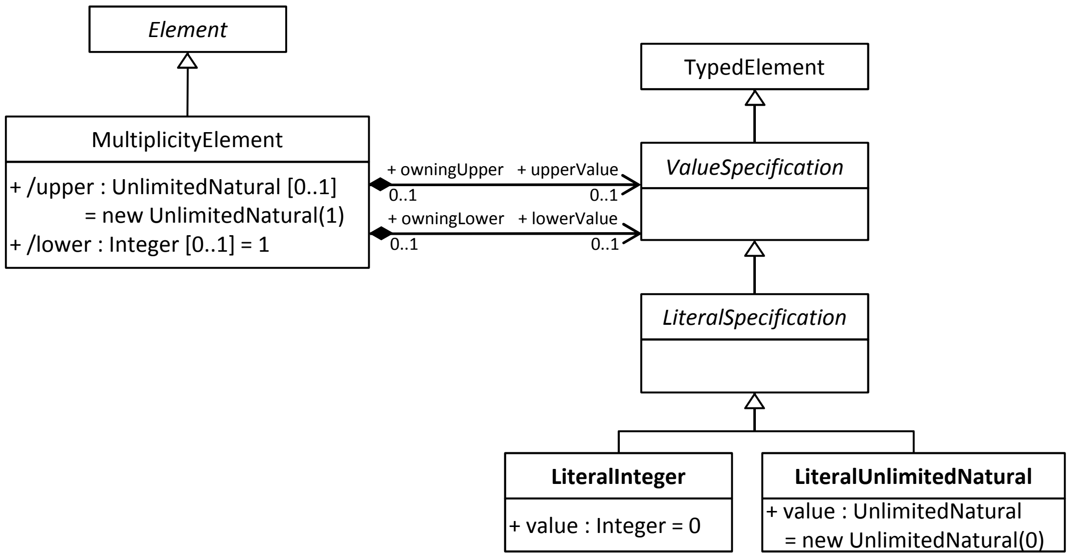


Figure 4.15: Class diagram elements supported by the plug-in - Multiplicity elements [20]

Semantic Object Diagram

The abstract syntax of the modeling concepts for semantic object diagrams supported by the implemented plug-in is depicted in the Figures 4.16, 4.17 and 4.18 in form of excerpts of the fUML metamodel.

What should be considered is that this object diagram is not the object diagram known from UML, but a semantic model whose concepts are used by the execution engine to describe the runtime instances during executing an activity. This is also the reason why there is no container element included in the metamodel that could be used as model element for a diagram. Because of this the metamodel was extended by the metaclasses `Package` and `PackageableElement` whereat the existing metaclasses `Object` and `Link` are subclasses of the abstract metaclass `PackageableElement`. This extension is depicted in Figure 4.16.

Figure 4.17 shows the metamodel excerpt that deals with the metaclasses `Object` and `Link`. Objects are instances of classes and because of this they maintain references to these classes. Similarly links are the instances of associations and therewith reference these associations. The classes and associations are defined in the class diagram which was described above. References to objects, i.e., references to instances of the metaclass `Object`, can be modeled using the modeling concept `Reference`. Instances of the metaclass `FeatureValue` represent instances of the structural features, i.e., the properties of classes, defined in the class diagram. Depending on the defined multiplicity of a property, a corresponding feature value has a certain number of values. Figure 4.18 shows the subclasses of the abstract metaclass `Value`, namely the metaclasses `StructuredValue` whose subclasses were already described and `PrimitiveValue`. An instance of the metaclass `PrimitiveValue` is the instance of a primitive type defined in the class diagram and therefore references this primitive type. Concrete subclasses of the metaclass `PrimitiveValue` are `BooleanValue`, `IntegerValue`, `StringValue` and `UnlimitedNaturalValue`.

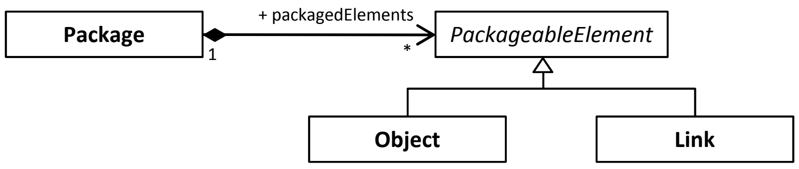


Figure 4.16: Semantic object diagram elements supported by the plug-in - Packageable elements

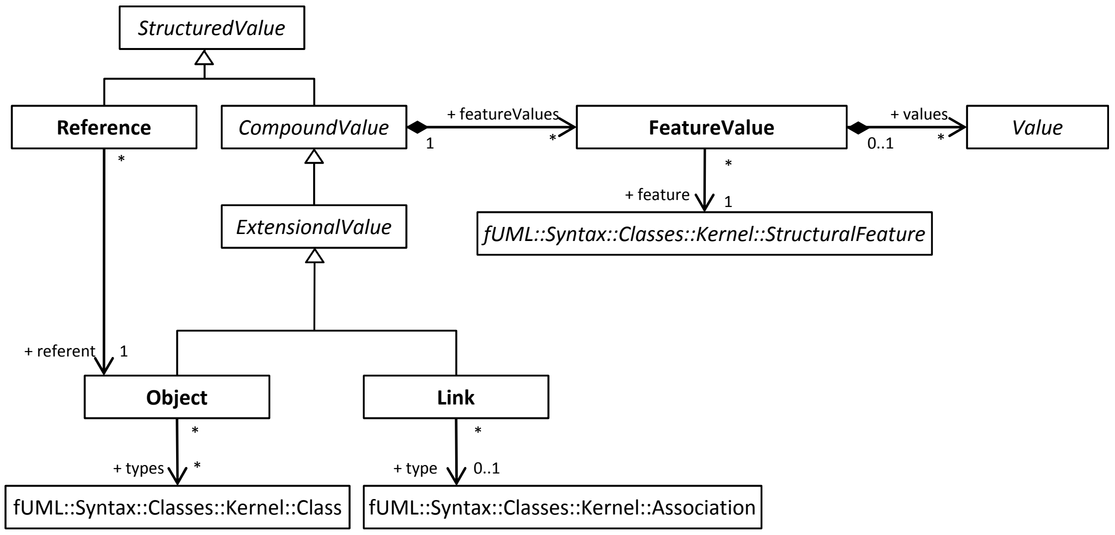


Figure 4.17: Semantic object diagram elements supported by the plug-in - Objects and links [20]

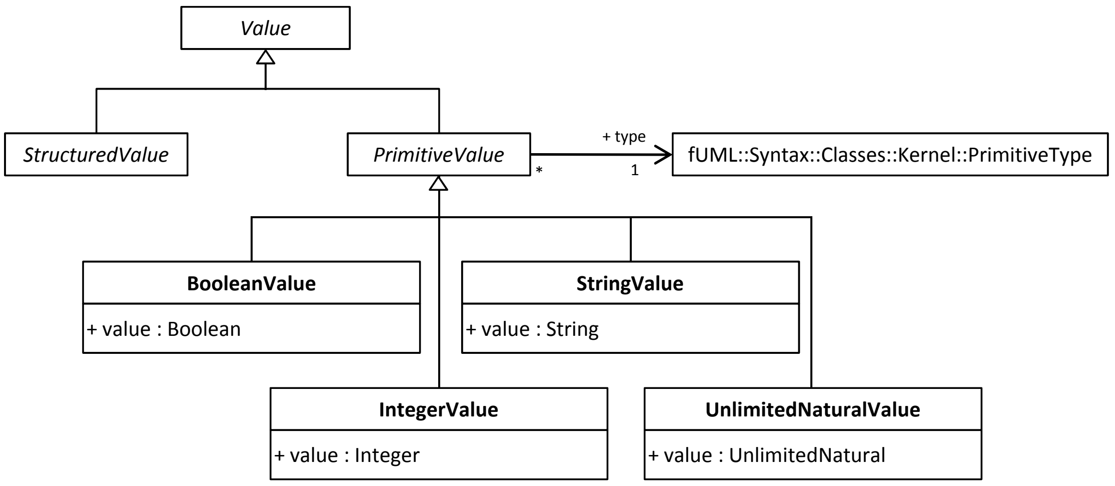


Figure 4.18: Semantic object diagram elements supported by the plug-in - Values [20]

Activity Diagram

An excerpt of the metamodel of the activity diagram is depicted in Figure 2.2 and was already explained in Chapter 2.3. Two restrictions were introduced in fUML regarding the presented elements:

- The metaclass `Pin` is abstract in fUML
- The relationship between the metaclasses `ActivityParameterNode` and `Parameter` is a 1:1 relationship in fUML instead of a 1:n relationship.

All modeling concepts depicted in Figure 2.2 are supported by the model interpreter prototype except of the control node `DecisionNode`. Worth mentioning is also the fact that each of these modeling concepts is a direct or indirect subclass of the metaclass `NamedElement` and therewith has an attribute `name`. In addition, `ObjectNode` is a subclass of `TypedElement` and instances therewith maintain references to their type.

The implemented model interpreter prototype further supports actions that are concerned with the manipulation of objects, their structural features, i.e., attributes, and links. These supported actions are now presented in more detail by presenting the relevant parts of the fUML metamodel and by summarizing the preconditions and postconditions of the actions as described in the fUML formal specification [20] and supported by the model interpreter prototype.

The metamodel excerpt of fUML concerned with the supported object actions is depicted in Figure 4.19. Supported are the actions `CreateObjectAction`, `DestroyObjectAction` and `ValueSpecificationAction`.

Create Object Action

Precondition The classifier which should be instantiated must be specified. This classifier has to be a class.

Postcondition An object with the given classifier as its type has been created and a reference to the object has been placed on the output pin of the action.

Destroy Object Action

Precondition A reference to the object which should be destroyed has to be provided at the input pin of the action.

Postcondition The referent object has been destroyed.

Value Specification Action

Precondition The value specification which should be evaluated must be provided.

Postcondition The value specification has been evaluated and the result has been placed on the output pin of the action.

The abstract syntax of the structural feature actions supported by the model interpreter prototype is depicted in Figure 4.20. The actions `ReadStructuralFeatureAction`, `Clear-StructuralFeatureAction`, `AddStructuralFeatureValueAction` and `Remove-StructuralFeatureValueAction` are supported.

Read Structural Feature Action

Precondition The structural feature which should be read has to be specified. Furthermore the object whose structural feature should be read has to be provided at the input pin of the action.

Postcondition The values of the specified structural feature of the given object have been retrieved and placed on the result output pin of the action.

Clear Structural Feature Action

Precondition The structural feature which should be cleared has to be specified and the object whose structural feature should be cleared must be provided at the input pin of the action.

Postcondition All values of the specified structural feature of the provided object have been cleared and the so modified object has been placed on the result output pin of the action.

Add Structural Feature Value Action

Precondition The structural feature for that a new value should be added has to be specified and the value which should be added has to be provided at the value input pin of the action. Furthermore the object for whose structural feature a value should be added has to be provided at the object input pin of the action.

Postcondition The provided new value has been added to the specified structural feature of the given object and the so modified object is provided at the result output pin of the action.

Remove Structural Feature Value Action

Precondition The structural feature for that a value should be removed has to be specified and the value which should be removed has to be provided at the value input pin of the action. In addition the object for whose structural feature a value should be removed has to be provided at the object input pin of the action.

Postcondition The provided value of the specified structural feature of the given object has been removed and the so modified object is provided at the result output pin of the action.

Figure 4.21 depicts the fUML metamodel excerpts that deal with the supported link actions namely `ReadLinkAction`, `CreateLinkAction`, `DestroyLinkAction` and `Clear-AssociationAction`.

Read Link Action

Precondition The data necessary to identify the link which should be read has to be provided:

- Data that identifies the link ends
- All end objects of the link except of the participating object that should be read

Postcondition The objects which are participating in the association and are not provided at the inputValue input pin have been placed on the result output pin of the action.

Create Link Action

Precondition The data necessary to specify the link which should be created has to be provided:

- Data that identifies the link
- All end objects of the link

Postcondition The specified link has been created.

Destroy Link Action

Precondition The data necessary to specify the link which should be destroyed has to be provided:

- Data that identifies the link ends
- All end objects of the link

Postcondition The specified link has been destroyed.

Clear Association Action

Precondition The object for which links should be destroyed has to be provided at the object input pin of the action. Also the association whose link should be destroyed if the provided object participates must be specified.

Postcondition All links of the specified association in which the provided object participates have been destroyed.

One additional action is supported by the prototypical interpreter called `OCLAction`. The made extensions of the fUML metamodel concerning the abstract syntax are depicted in Figure 4.22. The intension of the introduction of this action was to support the formulation of OCL queries that can be used to select objects with special properties. Also the semantic execution model had to be extended by a semantic activation visitor class called `OCLActionActivation` that implements the semantics of this new action.

OCL Action

Precondition The OCL query which should be evaluated has to be specified and the objects on which this query should be evaluated have to be provided at the input pin of the action.

Postcondition The query has been evaluated and the objects which conform to the query have been provided at the result output pin of the action.

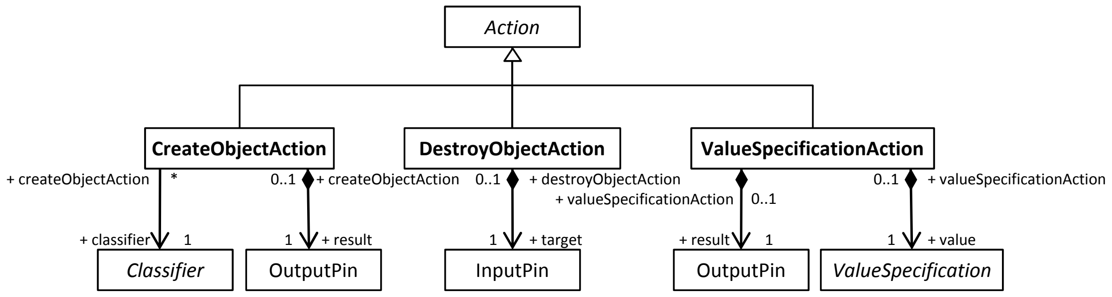
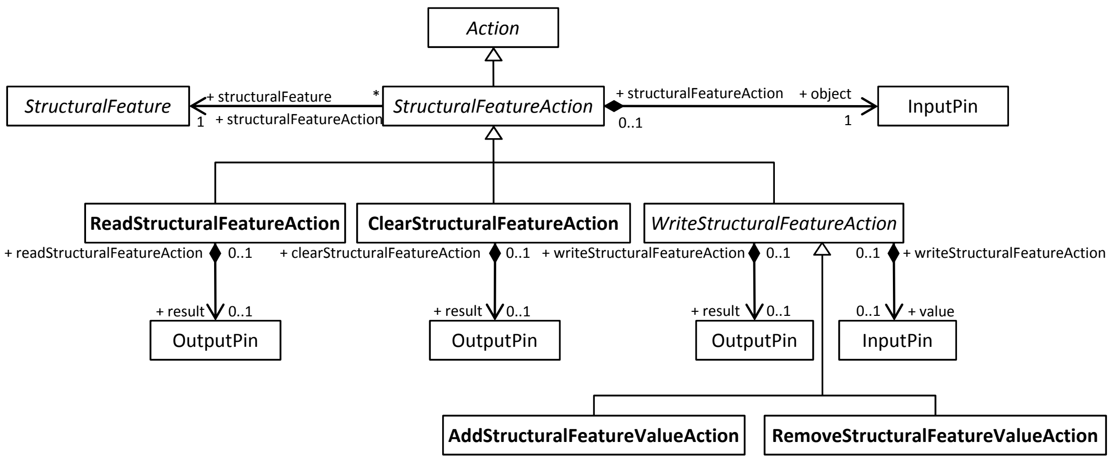


Figure 4.19: Activity diagram elements supported by the plug-in - Object actions [20]



Action
StructuralFeature
+ structuralFeature
1
*
+ structuralFeatureAction
StructuralFeatureAction
+ structuralFeatureAction
0..1
+ object
1
InputPin
ReadStructuralFeatureAction
ClearStructuralFeatureAction
WriteStructuralFeatureAction
+ readStructuralFeatureAction
0..1
+ clearStructuralFeatureAction
0..1
+ writeStructuralFeatureAction
0..1
0..1
+ writeStructuralFeatureAction
+ result
0..1
OutputPin
+ result
0..1
OutputPin
+ result
0..1
OutputPin
0..1
+ value
InputPin
AddStructuralFeatureValueAction
RemoveStructuralFeatureValueAction

Figure 4.20: Activity diagram elements supported by the plug-in - Structural feature actions [20]

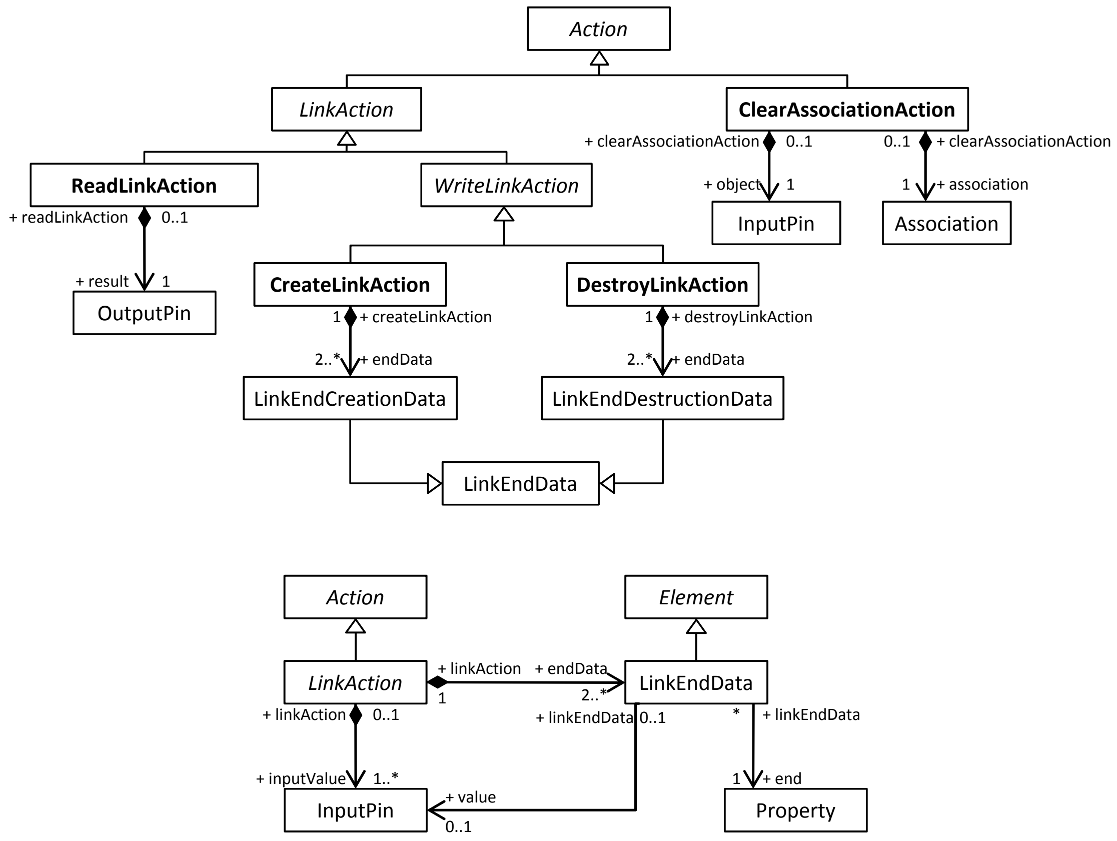


Figure 4.21: Activity diagram elements supported by the plug-in - Link actions [20]

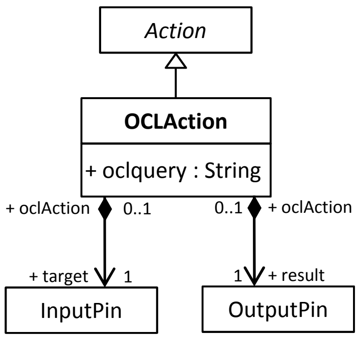


Figure 4.22: Activity diagram elements supported by the plug-in - OCL action

4.4 Possible Extensions of the Plug-In

The implemented model interpreter has to be considered as prototype because lots of additional functionalities would be necessary to provide a tool that supports an efficient and easy-to-use way to execute fUML models. This chapter presents ideas how the prototype could be extended to achieve this.

Extension of the supported modeling concepts

The implemented prototype only supports the execution of activity diagrams that model the manipulation of objects, their attribute values and links between these objects. So it supports only very few modeling concepts. The expansion of these modeling concepts is the most obvious possible extension of the model interpreter.

The modeling concepts could in fact be extended to support all UML modeling concepts of activity diagrams because as described in Chapter 3.1 and depicted in Figure 3.1, fUML is a computationally complete language for executing models and higher-level UML concepts can be translated into fUML.

Incorporation of a thread model

As described in Chapter 3.3, the prototype does not incorporate a threading model. The models are executed sequentially. This way of execution may not be the intended when modeling parallel execution paths using for instance fork nodes. Because of this the incorporation of a thread model would be a reasonable extension of the model interpreter.

Graphical editors

The implemented prototype only provides tree-based editors to define class diagrams, activity diagrams and semantic object diagrams. Graphical editors would provide a more convenient way to specify these models.

The Eclipse Graphical Modeling Project[1] (GMP) which is based on the Eclipse Modeling Framework could be used to implement graphical editors that enable the definition of models consisting of the supported UML or fUML modeling concepts in a graphical way.

Another way would be to use existing UML graphical editors of Eclipse like UML2 Tools which is part of the Eclipse Model Development Tools MDT Project[2]. The problem arising from the use of such an existing editor is that the available modeling concepts can't be restricted. I.e., the models would have to be tested for contained elements that are not supported by the model interpreter before the model can be executed.

Also existing modeling platforms like the Enterprise Architect[3] could serve as sources for executable UML models. But again the restriction of the supported modeling concepts for execution is an issue.

Debug visualization

Like the editor for creating activity diagrams provided by the prototype, also the view that depicts the progress of the debugging process displays activities using a tree view. The visualization of the debug progress could be much improved by displaying the activity diagram in concrete syntax, i.e., in graphical form.

Besides highlighting the currently executed activity node, also the visualization of the token flow would provide important information to the user, like which action could be executed in the next step, which action is waiting for a token to arrive etc.

Support of supplementary diagram types

The built model interpreter prototype enables the execution of activity diagrams according to the execution semantics defined in the fUML standard. Thereby the activity diagrams are based on class diagrams. But other UML diagram types could also be used as supplements. State machines and sequence diagrams constitute particular useful supplements.

[1]http://www.eclipse.org/modeling/gmp
[2]http://www.eclipse.org/modeling/mdt
[3]http://www.sparxsystems.com/products/ea/index.html

State machines could be used to define the lifecycle of objects of the classes which are defined in the class diagram. Therewith the states of objects and the possible state transitions could be defined and more important the feasible actions in each state could be specified. I.e., the state machine could be used to restrict the executable actions in each state of an object. During the execution of an activity diagram the state machine of the concerned object then has to be taken into consideration to determine if an action can be executed or not.

Sequence diagrams would also be a useful supplement for defining executable models. A sequence diagram could be used to describe the desired course of action, i.e., the execution flow, and it could therefore be used to define test cases. Therewith the expected execution flow could be defined beforehand the execution of the activity diagram and the actual execution flow could then be compared with the defined expected execution flow. Tests to ensure that certain execution flows do not appear could be accomplished in the same way.

The debugging functionality of the implemented model interpreter prototype provides a so-called trace view that presents the already executed actions in the chronology of their execution. The executed actions are simply displayed using a tree view. This trace could also be visualized more conveniently in a sequence diagram. The provision of a trace in the form of a sequence diagram would also be useful for the execution of activity diagrams and not only for the debugging.

Fault-tolerant execution of models

The implemented prototype does not explicitly cope with errors in models. The check for violations of OCL constraints defined in the fUML metamodel, i.e., syntactical errors, has to be manually triggered by invoking the “validate” functionality provided by the Eclipse EMF Validation Framework. Other restrictions defined by UML or fUML which can’t be expressed using OCL constraints are not addresses at all. The same is true for semantical errors. For instance the create object action has to have a result output pin where a reference to the instantiated object is placed. The validation framework would recognize the absence of such an output pin. But if this output pin has no outgoing object flow edge this probably constitutes a semantic error that can lead to an execution flow that was not intended and that is not handled by the model interpreter, although this error could have been detected very easily.

The following three extensions could lead to a more fault-tolerant model interpreter:

- **Automatic model validation**. The validation of the model to detect violations of OCL constraint defined in the metamodel could be triggered automatically before the execution of a model starts. Therewith execution errors caused by constraint violations could be eliminated.
- **Detection of potential errors**. Potential errors in the models like the described error of missing edges could be detected and warnings could be presented to the user before the

execution of this model. Therewith potential semantic errors could be prevented.

- **Handling of incomplete models**. A fault-tolerant model interpreter should also be able to execute incomplete models by enabling the user to provide the necessary but missing information like a missing guard condition during the model execution.

Backwards debugging

The debugging functionality of the model interpreter could be significant improved by providing the functionality of backwards debugging. This simply means that if the execution of an activity is paused at some point the user can not only move one step forward in the execution but also backwards. This simplifies the debugging process especially for large and complex models. Debugging is usually done by setting breakpoints at points where errors are suspected. But these points can only be guessed and therewith this process is time-consuming and error-prone. With backwards debugging it is possible to execute the model until an unintended behavior is detected and then to go backwards in order to determine where the error originated.

Analysis functionality

The incorporation of analysis functionality into the model interpreter prototype would also constitute a valuable extension. The detection of deadlocks or the identification of unreachable activity nodes are examples of reasonable checks that could be carried out for activity diagrams. The analysis support for Petri Nets could serve as a role model for this functionality. [1] is the first paper that is concerned with the analysis of fUML models. This approach suggests the formalization of fUML models into the process algebraic specification language CSP in order to detect deadlocks in these models.

Code generation

The provision of code generation functionality would complete the tool chain. With this extension platform-specific code could be generated out of the defined models that have been tested by executing or debugging them. To achieve the generation of platform-specific code, the platform-independent fUML models have to be enriched with platform-specific information and therefore have to be transformed into platform-specific models first. Out of these platform-specific models the code could then be generated.

CHAPTER 5

Discussion of the fUML Standard

The reason for the development of the fUML standard was the need for more precise models that could be executed to test and validate them. UML itself is not capable of meeting these demands because its action semantics is only informally described in English prose and is therewith neither precise nor complete. But with the emergence of the MDA approach in 2003 the need for precise models that could be run in order to validate them before they are deployed to the implementation platform gained in importance. In 2005 this need lead to the OMG's request for proposal for the Semantics of a Foundational Subset for Executable UML Models. In 2008 the resulting specification of the foundational UML (fUML) was adopted and in February 2011 the version 1.0 was released. Therewith a subset of UML got a precise standardized execution semantics. Of course, tools to execute models exist for years but each tool defines its own execution semantics and often uses its own action languages. Therewith the tools are not interoperable and models cannot be interchanged. By standardizing the execution semantics of UML models, this problem of interoperability of tools might become obsolete [27, 28].

By implementing a prototype for a model interpreter based on the fUML standard, that can be used to execute and debug UML activity diagrams, insight into the fUML standard was gained. This chapter is concerned with the experiences regarding fUML gained during working with this standard. Three main points of critique solidified throughout the development of the model interpreter prototype:

1. Executable models have to be too detailed and therewith rapidly exceed a reasonable size and are difficult to handle because a graphical syntax does not scale.

2. The form of provision of the fUML execution model is not optimal.

3. Model checking of the fUML execution engine is insufficient.

Executable models have to be too detailed

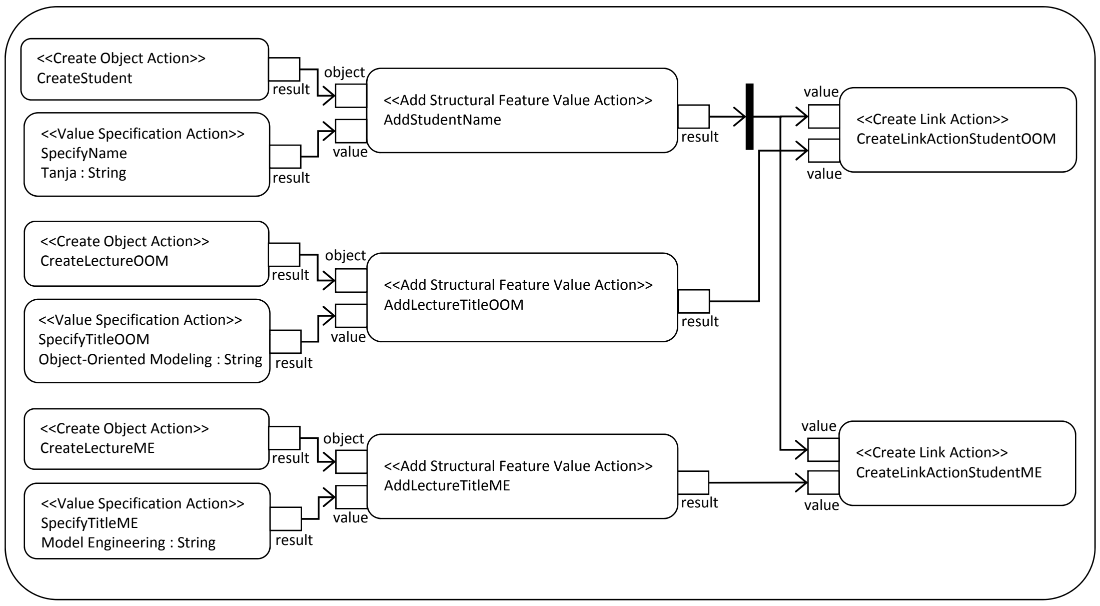
The problem with executable models in that they have to be described at a very low level of abstraction, i.e., the models have to be very detailed. In the context of fUML this means that the executable activity diagrams have to be defined at the lowest level of detail by using the supported primitive actions. This rapidly leads to very huge models that are by no means easy to handle. The creation of models on that low level is very time-consuming and error-prone and therewith not very efficient. Defining models on this detailed level resembles the action of programming and raises the question if it constitutes just another way of programming, namely "graphical programming" [27].

This problem can be exemplified using the example of an activity diagram depicted in Figure 4.10. Of course this is not a huge or unmanageable model but if it is considered that this activity diagram only creates three objects and links between them, it can be seen that it is fairly large and that more sophisticated functionality easily leads too very complex models. By opposing this exemplary activity diagram with the analogous Java code like in Figure 5.1, it can also be seen that the code is pretty easy to interpret whereas one needs a second glance at the model to understand the described behavior. This becomes even more apparent if it is considered that information necessary to understand the model is not displayed in the graphical model itself but in the properties of the model elements that have to be extra looked-up. Such a property is for instance the class that should be instantiated by a create object action or the property for that a value should be added by an add structural feature value action. This information was added in the graphical notation of the exemplary activity diagram to enable an easier understanding but it is usually not displayed by modeling tools but hidden in extra dialogs of the user interface.

This problem of the necessity for too detailed models in order to execute them was recognized by OMG and lead to the request for proposal for *Concrete Syntax for a UML Action Language* in 2008. The resulting specification of the *Action Language for fUML*[1] *(Alf)* was adopted in 2010 and is currently in the finalization phase. Alf is a Java-like textual notation for fUML models. Its execution semantics is specified by mapping the concrete syntax of Alf to the abstract syntax of fUML. Therewith fUML can be seen as the specification of a virtual machine for executing behavior that is defined using the Alf action language. The intention of Alf is to enable the specification of executable behavior of elements that are defined in a UML model that is represented graphically. An example would be the definition of the behavior executed in an operation of a classifier by using Alf. But Alf also includes a notation to represent structural modeling, and in this way a UML model can be entirely represented using Alf [17, 27].

This thesis does not deal with Alf in any way because the Beta 1 version of the specification was releases at a point of time at which the functionality of the model interpreter prototype was already determined.

[1]http://www.omg.org/spec/ALF



```java
Student student = new Student();
student.setName("Tanja");
Lecture lecture1 = new Lecture();
lecture1.setTitle("Object-Oriented Modeling");
Lecture lecture2 = new Lecture();
lecture2.setTitle("Model Engineering");

student.getLectures().add(lecture1);
student.getLectures().add(lecture2);
```

Figure 5.1: Comparison of activity diagram and analogous Java code

The form of provision of the fUML execution model is not optimal

The execution model of fUML specifies the execution semantics of the foundational UML subset in a formal and operational way using Java code. This means that the execution semantics of all modeling concepts included in the foundational UML subset as well as the fUML execution engine and environment are specified using Java code.

The execution model of fUML is contained in the formal specification document of the fUML standard and is organized according to the package structure of fUML. Within a package the contained classes are alphabetically arranged and the code of each class is provided. What such a class description looks like is illustrated in Figure 5.2 with the help of the specification of the semantic class `CreateObjectActionActivation` which is the activation visitor class that defines how a create object action has to be executed. What this example should illustrate is that it is difficult to get an overall picture of how the execution of an activity diagram actual works. For a reference guide the organization of the execution model in the formal specification of fUML is suitable, but to get an understanding of the model execution it is inappropriate.

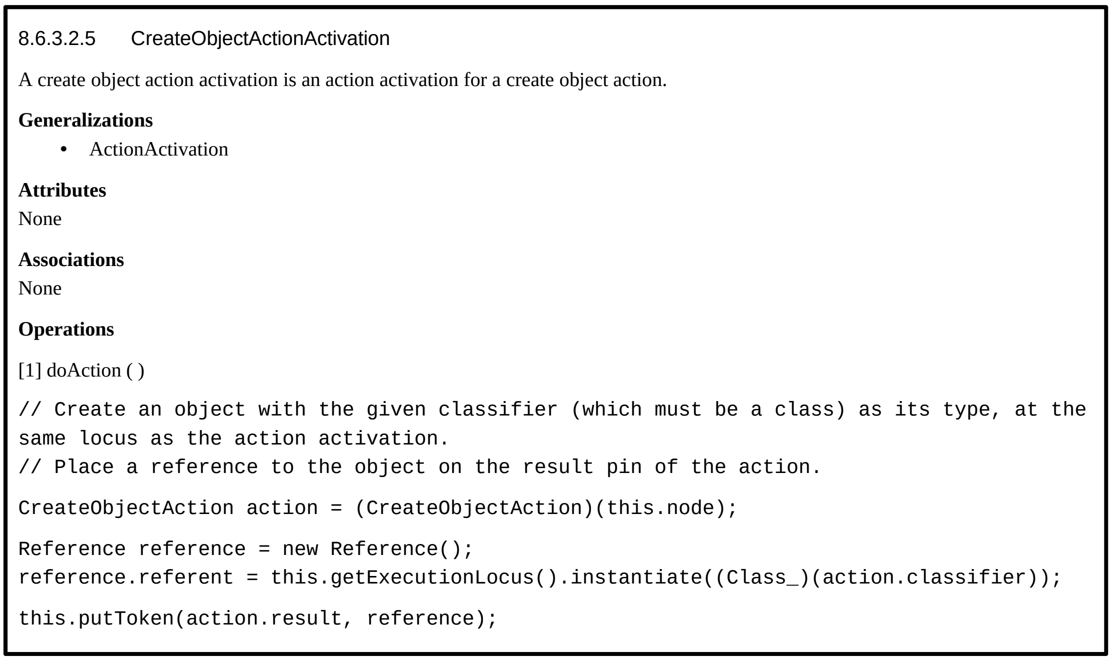
Another problem that is posed by the form of provision of the execution semantics is that lots of adoptions of the provided Java code are necessary in order to build a working execution engine for fUML activity diagrams. As described earlier, the fUML execution model for the built model interpreter prototype was implemented by generating the classes with their methods from the fUML metamodel provided by OMG and adding the Java code denoted in the formal specification. But by copying the code the work was not done because the code had to be adopted in order to even become compiled. For instance one annoying adoption was necessary because in the specification of the execution model it was assumed that the instance variables of the classes are declared as public and that therewith an external access to them is possible. This can be seen in the excerpt of the execution model specification provided in Figure 5.2. But according to the principle of encapsulation the instance variables of the syntactical classes were declared as private. Because of this, code that included direct access to instance variables was replaced by according invocations of getter or setter methods. Another cumbersome problem with the provided Java code was that during testing the execution engine lots of null pointer exceptions occurred due to missing variable initialization.

Despite the fact that the completion of the classes of the fUML execution model by copying the specified Java code of the fUML formal specification seems to be a pretty easy task, a high effort was necessary to really get a runnable fUML execution engine.

Therewith the form of provision of the execution model of fUML can be seen as suboptimal. An additional provision of the execution model in form of a runnable Java project would be very useful and would save efforts.

An open source reference implementation of fUML was built by Model Driven Solutions[2] and could have been used as starting basis for the model interpreter prototype built as part of this thesis. But it was decided to implement the prototype based on the standard documents provided by OMG to experience the whole process of building a conformant tool starting with the provided metamodel and formal specification of fUML.

[2]http://fuml.modeldriven.org


8.6.3.2.5 CreateObjectActionActivation

A create object action activation is an action activation for a create object action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Create an object with the given classifier (which must be a class) as its type, at the
same locus as the action activation.
// Place a reference to the object on the result pin of the action.

CreateObjectAction action = (CreateObjectAction)(this.node);

Reference reference = new Reference();
reference.referent = this.getExecutionLocus().instantiate((Class_)(action.classifier));

this.putToken(action.result, reference);
```

Figure 5.2: Exemplary class description of a semantic activation class of the fUML execution model specification [20, p. 237]

Model checking of the fUML execution engine is insufficient

During testing the model interpreter prototype by executing test models it was discovered that the executed models are not sufficiently checked by the execution engine. The following list enumerates detected examples of insufficient model checking:

1. **Instantiation of abstract classes**. It is possible to instantiate abstract classes by using the create object action. The constraint that abstract classes can't be instantiated is neither defined as OCL constraint in the metamodel, i.e., such a violation in an activity diagram can't be discovered by validating the model, nor is it checked by the activation visitor class `CreateObjectActionActivation` responsible for the execution of a create object action or by the class `Locus` that actually instantiates a class.

2. **Adding of feature values to objects for properties that do not belong to the class of the object**. The execution engine allows to add values to an object for a property that does not belong to the class of the object by using the add structural feature value action. An example using the class diagram depicted in Figure 4.3 would be to add a value for the property `Name` to an object of the type `Lecture` although this property belongs to the class `Student` and not to the class `Lecture`. Again this violation is neither detected by validating the model nor by the execution engine.

3. **Addition of an amount of feature values that exceeds the defined multiplicity interval of the property**. For every property of a class a multiplicity interval can be defined to constrain the amount of values for that property that can be added to an object. Unfortunately this multiplicity interval is not checked by the activation visitor class of the add structural feature value action and so more values can be added than permitted.

This list could be continued by looking at the constraints defined in the UML standard as well as in the fUML standard and analyzing if they are addresses by the fUML metamodel or execution model. But generally it can be said that no constraint that is defined in the standard documents and is not coped by OCL constraints in the metamodel, is checked in any other way. This leads to situations like described above where the execution engine executes behavior that is not allowed according to the standard or it leads to exceptions during the execution like null pointer exceptions or class cast exceptions.

This problem constitutes a severe weakness of fUML. To overcome it tool vendors have to extend the fUML metamodel and/or the fUML execution engine to check for constraint violations.

CHAPTER 6

Related Work

The fUML standard is the attempt of OMG to fulfill the need for a complete and precise execution semantics of UML. This need became compelling at the latest since the emergence of the MDA approach. It manifested in numerous proposals of semantic definitions for UML and approaches for executing models. Industrial as well as academic institutions have put efforts into the topic of model execution.

Examples for industrial tools supporting the execution of UML models are IBM's Rational Software Architect and the software Enterprise Architect of Sparx Systems. The *Rational Software Architect Simulation Toolkit* supports the execution of UML activity models, interaction models and state machines as well as the model-level debugging [22]. The Enterprise Architect plug-in called *Amuse* enables the execution and simulation of state machines [21].

At the moment there are no mature or established tools that allow users to define and execute UML models as specified in the fUML standard. The reason for this is that the fUML standard was only recently released in February 2011 and tool vendors will need some time to build conforming tools. What exists is a reference implementation of fUML implemented by Model Driven Solutions and also some research was done concerning fUML. The fUML reference implementation as well as the paper [14] are presented in more detail in Chapter 6.1. [14] is highly related to this thesis at hand because it describes a tool chain for fUML that supports similar functionality as the model interpreter prototype built in course of this thesis and these two approaches are directly compared with each other for this reason.

Only few papers exist that deal with the execution of UML 2 activity diagrams. Two of them are briefly presented in Chapter 6.2 because they provided interesting ideas for this thesis. More approaches exist that deal with the execution of UML state machines. One prominent example is *Executable UML (xUML)*. xUML is a subset of UML for which the execution semantics is precisely specified. Executable models are defined using modeling concepts of class diagrams as well as of state machines. The behavior of the operations of the defined classes as well as the actions of the states in a state machine are specified using a UML Action Language that depends on the chosen xUML tool [24].

At this point it has to be mentioned that the action semantics of UML and an UML action language were already an issue in 1998 when a request for proposal concerning this subject was issued by OMG and lead to an expanded action metamodel in UML 1.5. This action metamodel had a strong influence on the abstract syntax for actions in UML 2.0 but the action semantics was neither in UML 1.5 nor in UML 2.0 formally specified and also no action language was developed until the specification of Alf [28].

Chapter 6.3 presents the Fujaba project and the Kermeta project which are also related to this thesis. The Fujaba Tool Suite enables the generation of code from UML class diagrams and a combination of UML activity diagrams and a graph-transformation language. Fujaba also provides a test environment where a modeled system can be simulated. Kermeta is a metamodeling language that does not only support the specification of the structure of metamodels but also the definition of its semantics.

6.1 Work concerning the Execution of fUML Models

fUML Reference Implementation

Model Driven Solutions developed an open source fUML reference implementation in 2008. This reference implementation is currently in version 0.4.1 which was released in February 2011 and it is conformant to the fUML standard version 1.0 [15].

An UML model in XMI format has to be provided as input to the fUML reference implementation in order to execute it. The provided model is then executed and after the execution is finished the execution trace is provided as output [15].

The objective of building this reference implementation was to encourage tool vendors to build conformant tools by providing them a reference of how to build such a tool. By this means it also serves as aid for evaluating implementations of tool vendors in respect of conformance with the fUML standard. Declared objective is also the enhancement of the reference implementation by proving additional functionality like a debugging functionality and the support of the evaluation and evolution of the fUML standard itself [15].

Tool Chain for fUML

A research group at the Department of Computer Science of the Babeş-Bolyai University in Romania developed a tool chain for constructing and testing executable UML models according to the fUML standard, as well as for generating code from these models [14].

The tool chain was developed based on the Eclipse Modeling Framework. Especially interesting is the fact that this research group introduced its own action language for fUML in [13] because they noticed how hard it is to create executable UML activity diagram due to the necessary low abstraction level and OMG had only issued an request for proposal for a concrete syntax for an action language based on fUML at the time at which the tool chain was developed.

The tool chain supports the following three steps of constructing and testing models by providing appropriate tools [14]:

1. **Model creation**. In the first step the executable fUML model is created consisting of the following two components:

    a) **Class diagram**. First a class diagram has to be constructed that defines the structural aspects of the model. The tool chain designates the usage of the Eclipse UML Class Diagram Editor that is part of the Eclipse UML2 Tools[1].

    b) **Activity diagram**. After the creation of the class diagram the behavior of the operations of the defined classes has to be specified. As mentioned before, this behavior is defined using the action language that was developed by the research group. The syntax of this action language was chosen to resemble the syntax of modern programming languages like Java in order to enable an easy usage of this language and to reduce the effort for learning it. A textual editor for this action language was built using the Xtext project of Eclipse[2]. This editor is integrated with the class diagram editor so that if an operation of a class is selected, this editor can be opened and used to specify the behavior of this operation. After the specified behavior is saved, the textual representation of this behavior is automatically converted into an activity diagram conformal to fUML.

2. **Model execution**. The second step comprises the execution of the created model, i.e., the execution of the behavior specified using the action language. To support this step the above presented reference implementation of fUML built by Model Driven Solutions was integrated in the tool chain.

3. **Code generation**. After testing the created executable model by executing it, code can be generated. For this step an implementation of the OMG MOF Model to Text Language MOFM2T[3] called Acceleo[4] was used. The templates necessary to generate code from the activity diagrams were implemented by the research group.

[1]http://www.eclipse.org/modeling/mdt#uml2tools
[2]http://www.eclipse.org/Xtext
[3]http://www.omg.org/spec/MOFM2T/Current
[4]http://www.eclipse.org/modeling/m2t#acceleo

The presented tool chain and the model interpreter prototype built in the course of this thesis both have the same objective: To provide a tool that can be used to create and execute UML models as specified in the new fUML standard. Another similarity is that they are both built on top of the Eclipse Modeling Framework. But on closer examination of the functionality provided by the tools it can be seen that they work in different ways. The following list enumerates the major differences:

- **Class diagram editor**. The presented tool chain of the Romanian research group suggests the usage of the graphical Eclipse class diagram editor which is part of the UML2 Tools. The usage of this editor leads to the problem that the users have to keep in mind which modeling concepts are supported by fUML and which are not and it is up to them to use only supported elements. Another downside is that additional constraints included in the fUML metamodel also can't be addressed. The model interpreter prototype built for this thesis overcomes these problems by providing an editor that supports only fUML modeling concepts. But because this editor is only a tree-based editor and not a graphical one it is also suboptimal and the provision of an equivalent graphical editor would of course lead to an increase of usability.

- **Activity diagram editor**. The extra developed action language for fUML and the corresponding textual editor of the tool chain implemented by Lazăr et al. constitute a main difference. The built model interpreter prototype does not support any action language but provides a tree-based editor for defining fUML compliant activity diagrams.

- **Execution engine**. Lazăr et al. integrated the fUML reference implementation provided by Model Driven Solutions whereat the execution engine of the built prototype was implemented from scratch based on the fUML metamodel and formal specification.

- **Debugging of models**. The tool chain described in [14] does not provide a debugging functionality.

- **Code generation**. A main component of the tool chain introduced in [14] is built up by the provided code generation functionality which is not included in the functions of the built model interpreter prototype.

Table 6.1 summarizes the similarities and differences of the tool chain provided by Lazăr et al. and the model interpreter prototype built in course of this thesis.

Criteria	*Tool Chain by Lazăr et al.*	*Model Interpreter Prototype*
Platform	Eclipse Modeling Framework	Eclipse Modeling Framework
Modeling language	fUML	fUML
Execution semantics	fUML	fUML
Execution engine	fUML reference implementation by Model Driven Solutions	implemented based on fUML metamodel and formal specification
Functionality		
Creation of models	supported	supported
Execution of models	supported	supported
Debugging of models	not supported	supported
Code generation	supported	not supported
Editors		
Class diagram	graphical Eclipse class diagram editor of UML2 Tools	tree-based editor
Activity diagram	textual editor integrated with class diagram editor	tree-based editor
Action language	own action language, automatic transformation into fUML compliant activity diagram	no action language
Object diagram	not supported	tree-based editor for specifying expected output of model execution, actual output of the execution is saved in the same format

Table 6.1: Comparison of the tool chain provided by Lazăr et al. and the model interpreter prototype built in course of this thesis

6.2 Work concerning the Execution of UML Activity Diagrams

Approaches for the execution of UML activity diagram already existed prior to the development of fUML. Two approaches which were developed in the UML 2 Semantics Project provided important inputs to this thesis and are for this reason briefly presented.

UML 2 Semantics Project

The UML 2 Semantics Project started in 2005 and was an international collaboration with the main objective of developing a mathematically formalized semantics definition for the UML 2 standard. The semantics foundation of this semantics definition developed in this project is called the *System Model*. Participants of this project included IBM, Queen's University, Technical University of Munich and Technical University of Braunschweig. At the time the project expired some of the documents constituting the semantics definition were only available as draft versions and remained unfinished. The working documents can still be found at [29].

The UML 2 Semantics Project also aimed at defining a UML virtual machine for executing UML models and two different sub-groups dealt with this goal leading to two different solutions. The group at IBM implemented a generic model execution engine for simulating any kind of models (even non-UML models) and they built a UML simulator for activity diagrams on top of it. At the same time researchers at the Queen's University developed an execution and analysis engine for UML activity diagrams based on the System Model. This engine was designed to be extensible in order to develop a comprehensive UML virtual machine [2].

UML Simulator Based on a Generic Model Execution Engine

The sub-group of the UML 2 Semantics Project at the research lab of IBM Haifa developed an architecture for implementing a generic model execution engine that can be used to simulate models. The most interesting aspect of this execution engine is that it is designed to be *generic* in order to support the simulation of any kind of behavioral models regardless of the used modeling language. Like fUML also this approach designates that the behavior of the modeling concepts, i.e., the execution semantics of the modeling language, is defined using Java code. To achieve the *extensibility* to support any modeling language this execution engine provides a standard set of execution mechanisms that can be re-used for the semantics specification of numerous modeling languages [12].

Observability and controllability are also two important aspects of the generic model execution engine built by the research lab of IBM Haifa. The execution of an activity is *observable* by providing well-defined interfaces for behavioral exploration tools like debuggers. *Controllability* means that the execution of a model is controllable [12].

On top of this architecture of a generic model execution engine a *UML simulator* was implemented by the research lab in form of an extension to the IBM Rational Software Architect which is based on the Eclipse software framework. The architecture of this UML simulator is depicted in Figure 6.1. The UML simulator supports the execution and debugging of activities that were defined using the Rational Software Architect and Java is supported as action language. For debugging UML activities this UML simulator provides very similar functionality as the model interpreter prototype built in the course of this thesis: It enables the user to execute the activity step-by-step and it is also possible to define breakpoints for activity nodes so that the execution can be carried out until a breakpoint is reached. Also the current state of the execution is visualized even in the graphical activity diagram. This visualization includes the information which activity nodes are ready to be executed, which activity edges are passing tokens and which activity nodes are providing tokens. The user also has the possibility to observe the attribute values of the existing objects. An interesting functionality of this UML model simulator is that it enables the user to invoke operations of objects and to create and destroy objects during the execution of an activity [12].

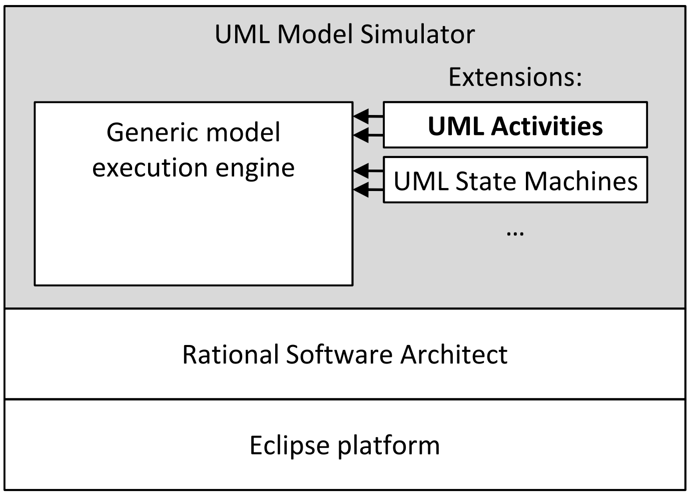


Figure 6.1: Architecture of the UML model simulator built by the research lab at IBM Haifa [12]

Interpreter for UML Activity Diagrams Based on the System Model

Based on the System Model developed in the UML 2 Semantics project an interpreter for UML actions and activities called *ACTi* was implemented by a sub-group of the project members. The architecture of this interpreter was designed to enable its extension in order to allow the execution of other behavioral formalisms like state machines aiming at developing a comprehensive UML virtual machine [3].

ACTi is a Java program that executes UML activities and provides analysis capabilities. The activity that should be executed has to be provided in a textual form called *Activity Diagram Linear Form (ADLF)*. Besides containing primitive actions the activity can also consist of self-defined actions whose behavior has to be specified using Java code. Additionally a simplified form of a class diagram has to be provided as input also in a defined textual representation. Optional is the provision of a kind of object diagram that describes in a textual way which objects exist at the beginning of the activity execution [3].

Very interesting is the tracing and analyzing functionality of ACTi. After executing the provided activity the interpreter provides the *trace* of the execution that not only includes the chronology of the executed activity nodes but also more detailed information like for instance how tokens were offered and passed. The trace information can also be exported and provided to a third-party tool for visualization purpose. Furthermore ACTi also carries out *analysis* during the execution of an activity and afterwards. Three kinds of analysis are provided by ACTi [3]:

- **Path analysis**. For this type of analysis the user has to specify information about which execution paths are desired and which are not. An example is the check for desired nodes where the user defines which activity nodes should be executed and ACTi checks if all of these nodes actually were executed.
- **Analysis of unused tokens or offers**. This analysis enables the detection of deadlocks by checking if after the execution of an activity is finished tokens remained unused or unoffered in the activity.
- **Sanity checks**. During execution so-called sanity checks are carried out that aim at detecting errors in the model. For instance activity edges that lead to an initial node are detected by the interpreter during the execution of the activity and a warning is presented to the user.

6.3 Other Related Work

Fujaba Tool Suite

The *Fujaba Tool Suite* is an open source tool for model-based software engineering and re-engineering that was developed at the University of Paderborn in 1997. By now the Fujaba Tool Suite is further developed by research groups at universities all over Germany and other countries. Besides supporting model-based software engineering and re-engineering, extensions of the Fujaba Tool Suite exist that provide additional functionality and support features for a variety of application domains. For example extensions exist for reverse engineering and for validation and verification of embedded real-time systems [8].

In the Fujaba Tool Suite UML class diagrams, UML activity diagrams and a graph-transformation language called *Story Patterns* are used to specify the structural and behavioral aspects of a software system. From these specifications the Fujaba Tool Suite generates executable Java code [8].

UML class diagrams are used to describe the structural aspects of a software system. To describe the behavioral aspects so-called *Story Diagrams* are used in order to specify the behavior of the operations of the defined classes. A Story Diagram is a combination of a UML activity diagram and Story Patterns. A Story Pattern is a graph transformation rule that describes the modification of objects in a system. Modeling concepts of UML activity diagrams are used in a Story Diagram to express the control flow among these graph transformation rules. A Story Pattern is depicted in a graphical notation that resembles UML object diagrams. The defined object structure of such a Story pattern constitutes the left-hand side of the graph transformation rule and the creation and destruction of objects and links in this object structure denotes the right-hand side [5].

From the class diagram and the Story Diagrams an executable program is generated. The Fujaba Tool Suite provides a test environment called *Dynamic Object Browsing System* that can be used to simulate the execution of this generated program. During the simulation the dynamic behavior of that program is depicted by means of an object diagram that visualizes the manipulation of the existing objects in the system. In course of this simulation the user can invoke the operations of the defined classes whose behavior is defined in a Story Diagram. The user can also instantiate and destroy objects [9].

The Fujaba Tool Suite provides very similar functionality as the model interpreter prototype built in course of this thesis. Both can be used to test models that specify how objects are manipulated in a system although the Fujaba Tool Suite only supports a combination of UML activity diagrams and a graph transformation language instead of pure UML activity diagram to specify the behavior of the system under development.

Kermeta

Kermeta is like MOF a metamodeling language. Unlike MOF Kermeta does not only enable the specification of the structure of metamodels but also allows the definition of the dynamic semantics of a metamodel [16, 23].

Kermeta is compliant with the metamodeling languages EMOF of OMG and Ecore of Eclipse. Besides supporting the definition of the structure of a metamodel, Kermeta also provides an action language that can be used to define the operational semantics of a metamodel in its operations. Kermeta can be used to implement metamodeling languages, action languages, constraint languages and transformation languages. It incorporates concepts from the metamodeling language MOF, the constraint language OCL and the model transformation language QVT. Kermeta has an imperative syntax, it is object-oriented, model-oriented and aspect-oriented [4].

Kermeta provides an action language for MOF models and it can be seen as an extension of MOF. This means that one can specify the structural aspects of a metamodel using EMOF and add the behavioral specification afterwards using Kermeta [4]. This is visualized in Figure 6.2.

The Kermeta development environment is fully integrated with Eclipse and it provides an interpreter and debugger that allows to run a metamodel [23].

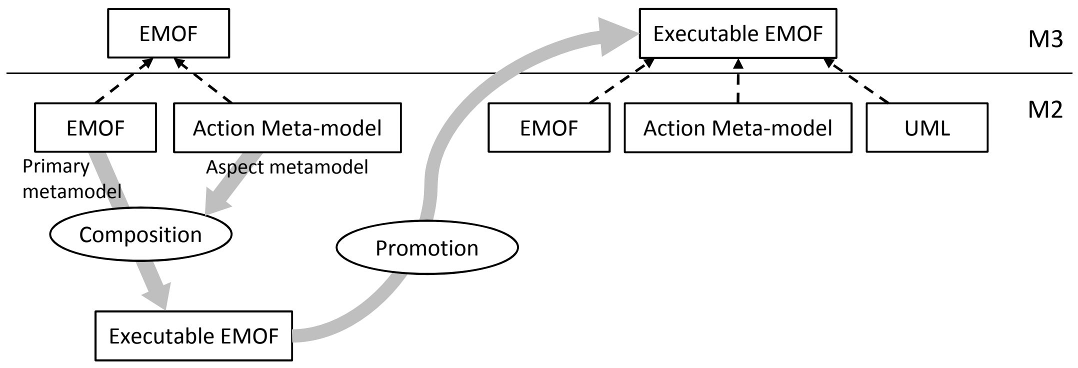


Figure 6.2: Architecture of Kermeta [4]

The model interpreter prototype built for this thesis was implemented using Java. Kermeta constitutes an alternative technology for implementing a tool that enables the execution and debugging of UML models according to the fUML standard. With Kermeta the metamodel of fUML, which is provided by OMG using MOF, can be enhanced by the execution semantics using Kermeta's Action Language. This approach doesn't require the code generation from the metamodel and the structural as well as the behavioral aspects of fUML can be described together in the fUML metamodel. [6] proposes an implementation of fUML using Kermeta.

CHAPTER 7

Summary and Future Work

7.1 Summary

The need for executable models that can be tested and validated by executing them arose with the emergence of the Model-Driven Development (MDD) paradigm. OMG's MDD approach known as the Model Driven Architecture (MDA) suggests the usage of the Unified Modeling Language (UML) to create platform-independent models. UML is also the de facto standard for modeling software systems. This thesis pointed out that UML models are not executable per se because UML has no precise and complete execution semantics. Because of this shortcoming the OMG released the new foundational UML (fUML) standard in February 2011. This standard defines the precise execution semantics of a subset of UML 2, the so-called foundational UML subset. The semantics of every modeling concept included in the foundational UML subset as well as an execution engine for compliant models is defined by the standard using Java code.

Based on the new fUML standard, a prototypical model interpreter was implemented as main outcome of this thesis. The aim was to enable the execution and debugging of UML activity diagrams that model the manipulation of objects and links in a system. So the model interpreter prototype supports activity diagrams that specify how objects are created and destroyed, how attribute values of objects are set and unset and how links between objects are created and destroyed. Therewith the supported modeling-concepts of fUML were restricted to object actions, structural feature actions and link actions.

The model interpreter prototype was built by generating the code from the fUML metamodel provided by OMG and implementing the methods of the semantic classes according to the execution semantics defined in the formal specification of fUML. To accomplish this, the Eclipse Modeling Framework (EMF) was used because it offers all functionality necessary to generate code from a metamodel that is defined in a MOF-based format. Based on the implemented model interpreter, an Eclipse plug-in was developed that provides the user interface for executing and debugging models. This plug-in provides tree-based editors for defining class diagrams that specify the static structure of objects and relationships between them, activity diagrams that define the manipulation of objects and links and semantic object diagrams that specify the expected output of the execution of an activity diagram. Activity diagrams can then be executed and debugged and the actual output can be saved in form of a semantic object diagram that can then be compared with the defined expected output. The debugging feature of the plug-in provides functionality that resembles the functionality known from the debugging of code: The model can be executed stepwise, breakpoints can be set to simplify the debugging process, the progress of the debugging is displayed as well as the existing runtime objects and a trace that depicts the chronology of the already executed activity nodes.

This thesis also suggested extensions of the model interpreter prototype and the Eclipse plug-in in order to provide an enhanced functionality for executing and debugging models in order to test and validate them. The provision of graphical editors, the visualization of the debugging progress in graphically displayed activity diagrams and analysis functionality are three examples of possible extensions.

By implementing the model interpreter prototype an insight was gained into the fUML standard and three main issues were experienced. The first of these issues is the fact that executable models have to be very detailed by using only primitive UML actions what leads to models that quickly exceed a reasonable size and become hard to handle. A new standard of OMG called Action Language for fUML (Alf) that is currently in the finalization phase deals with exactly this problem. Another point of critique is the form of provision of the fUML execution model that suits as a reference guide but is inappropriate to give an overall understanding of how models should be executed. Another shortcoming of the fUML standard that was detected during implementing the prototype is that the checking of the models that shall be executed is insufficient. In general it can be said that all constraints defined in UML and in fUML which are not addressed in the fUML metamodel by means of OCL constraints are also not handled by the execution engine. This leads to the execution of behavior that is not allowed according to the constraints defined in the standard documents or to runtime errors.

Tools for executing UML models were implemented prior to the development of fUML. Examples are the Amuse plug-in for the Enterprise Architect of Sparx System or the Rational Software Architect Simulation Toolkit of IBM. Because UML has no precise and complete execution semantics leaving room for interpretation, each tool implements its own execution semantics and these semantics are typically not exactly the same. By standardizing the semantics of UML in the fUML standard, the first step to overcome this problem of interoperability is done. At the moment there is no established or mature tool that allows users to define and execute UML models as specified in the fUML standard. The reason for this is that the fUML standard was only recently released in February 2011 and tool vendors will need some time to build conformal tools.

7.2 Future Work

In order to provide comprehensive functionality for executing and debugging UML models the prototypical model interpreter that was built in this thesis indeed has to be extended. The following extensions are the most interesting ones:

- **Modeling concepts and diagram types**. The built model interpreter prototype restricts the UML modeling concepts very much and only supports activity diagrams that model the manipulation of objects and links. The expansion of the supported modeling concepts for UML class diagrams and activity diagrams as well as the support of supplementary diagram types like sequence diagrams and state machines constitute a valuable extension.

- **Parallel execution**. The support of the parallel execution of an activity diagram is also a reasonable extension of the prototypical model interpreter. Currently the prototype executes activity diagrams sequentially and does not incorporate a thread model.

- **Graphical editors and debugging process visualization**. For creating class diagrams and activity diagrams the implemented model interpreter prototype provides only tree-based editors and also the progress of the debugging process is visualized in a tree-based view. The provision of graphical editors as well as the visualization of the debugging process in graphical displayed models would lead to an enhanced usability of the model execution tool.

- **Fault-tolerant model execution**. The automatic validation of models for violations of constraint defined in the fUML metamodel, plausibility-checks and the support of the execution of incomplete models by enable the user to provide the missing information during the execution process are features that would also enhance the usability of the model interpreter.

List of Abbreviations

Alf	Action Language for fUML
CMOF	Complete Meta Object Facility
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
fUML	Foundational UML
MDA	Model Driven Architecture
MDD	Model-Driven Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
UML	Unified Modeling Language
XMI	Extensible Markup Language

Bibliography

[1] I. Abdelhalim, J. Sharp, S. Schneider, and H. Treharne. Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 371–387. Springer Berlin / Heidelberg, 2010.

[2] M. Broy, M. L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In *MoDELS 2006 Workshops*, volume 4364 of *LNCS*, pages 318–323. Springer, 2006.

[3] M. L. Crane and J. Dingel. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '08, pages 8:96–8:110, New York, NY, USA, 2008. ACM.

[4] Z. Drey, C. Faucher, F. Fleurey, V. Mahé, and D. Vojtisek. *Kermeta Language Reference Manual*, November 2010.

[5] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, 1998.

[6] A. Fonseca. Coping with Modular Modelling in fUML. Master's thesis, University of Rennes 1, 2010.

[7] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[8] Fujaba project web page. http://www.fujaba.de. Accessed: 2011-04-12.

[9] Software Engineering Group. *FUJABA Documentation Guided Tour Version 0.3*. University Paderborn, June 2002.

[10] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML @ Work, Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, 3. edition, 2005 (in German).

[11] H. T. Jung and S. H. Joo. Transformation of an activity model into a Colored Petri Net model. In *Trendz in Information Sciences Computing (TISC), 2010*, pages 32 –37, 2010.

[12] A. Kirshin, D. Dotan, and A. Hartman. A UML Simulator Based on a Generic Model Execution Engine. In *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 324–326. Springer Berlin / Heidelberg, 2007.

[13] C.-L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, and I.-G. Czibula. Using a fUML Action Language to Construct UML Models. In *Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '09, pages 93–101, Washington, DC, USA, 2009. IEEE Computer Society.

[14] C.-L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, and I.-G. Czibula. Tool Support for fUML Models. *International Journal of Computers Communications & Control*, 5(5):775–782, 2010.

[15] ModelDriven.org. http://portal.modeldriven.org/project/foundationalUML. Accessed: 2011-04-03.

[16] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg, 2005.

[17] Object Management Group. Action Language for Foundational UML (Alf), October 2010.

[18] Object Management Group. OMG Unified Modeling Language Infrastructure Specification, Version 2.3, May 2010.

[19] Object Management Group. OMG Unified Modeling Language Superstructure Specification, Version 2.3, May 2010.

[20] Object Management Group. Semantics of a Foundational Subset for Executable UML Models, March 2010.

[21] Amuse product web page. http://www.lieberlieber.com/unser-angebot/amuse.html. Accessed: 2011-04-05.

[22] Rational Software Architect Simulation Toolkit product web page. http://www-01.ibm.com/software/rational/products/swarchitect/simulation. Accessed: 2011-04-05.

[23] Kermeta project web page. http://kermeta.org. Accessed: 2011-04-12.

[24] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML.* Cambridge University Press, 2004.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[26] B. V. Selic. On the Semantic Foundations of Standard UML 2.0. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 75–76. Springer Berlin / Heidelberg, 2004.

[27] Software Modeling Blog. http://modeling-languages.com/blog/content/new-executable-uml-standards-fuml-and-alf. Accessed: 2011-04-02.

[28] Software Modeling Blog. http://modeling-languages.com/blog/content/uml-action-language-omg-journey. Accessed: 2011-04-02.

[29] UML 2 semantics project web page. http://www.cs.queensu.ca/ stl/internal/uml2. Accessed: 2011-04-05.