UPTEC IT 10 019
Examensarbete 30 hp
September 2010

An Implementation of an Execution Engine for the Foundational UML Subset

Magnus Rundlöf

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

An Implementation of an Execution Engine for the Foundational UML Subset

Magnus Rundlöf

This thesis examines an OMG proposal for a tool-chain for working with Executable UML. The proposal defines a subset of UML, called fUML, with well-defined semantics and an Execution Model. The goal of this thesis is to understand whether fUML models can be useful in commercial user interface development. The idea is that an executable model (1) receives input from the data layer, which triggers an execution of the model, (2) transforms the data according to the model, i.e., executes the model on the given input, and (3) pushes the result to the presentation layer, or another part of the program that listens for the transformed data. We use the Eclipse Modeling Framework (EMF) to implement an execution engine for fUML models and show a full example, including constructing and executing a model. We conclude that Executable UML is not mature enough to be used instead of traditional code in commercial quality products. Firstly, the model construction tools we have used can not compete with modern development environments for code and we find it cumbersome to work with the models. Secondly, UML lacks important primitive types, e.g., real numbers, which forces the execution engine to work with strings, which in turn might cause performance problems. We also find it difficult to mix vendors in the tool-chain, for example, a user might want to buy a modeling tool to do the actual modeling, but use an open-source execution engine. This is difficult because the format on which models are stored is not well-defined. We suggest a few improvements to the Graphical Modeling Framework (GMF) UML2 Editor. These improvements together with an extension of the UML primitive types and the development of a model library, would make fUML more useful in practice.

Handledare: Lars Millberg
Ämnesgranskare: Sven-Olof Nyström
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT10 019
Tryckt av: Reprocentralen ITC

Preface

This is my Masters thesis in Information Technology at the Department of Information Technology, Uppsala University. The actual work was done at the Naval Systems Division of Saab Systems, Saab AB in Järfälla, Stockholm. I would like to thank Lars Millberg, who has been my supervisor at Saab, and Sven-Olof Nyström who was my examiner at Uppsala University. I also want to thank Rickard Westberg for hiring me to do this project, and Mattias Holmqvist to keep me company.

Magnus Rundlöf

September 2, 2010, Stockholm

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

EMF Eclipse Modeling Framework

fUML Foundational UML Subset

GMF Graphical Modeling Framework

HMI Human Machine Interface

IDE Integrated Development Environment

JAR Java Archive

JIT Just-In-Time

JVM Java Virtual Machine

MDA Model Driven Architecture

OMG Object Management Group

OOAD Object-Oriented Analysis and Design

OSGi Open Services Gateway initiative

P2C Polar-to-Cartesian

UML Unified Modeling Language

XMI XML Metadata Interchange

XML eXtensible Markup Language

Chapter 1

Introduction

1.1 Terminology

When we write executable UML, using lower-case e, we refer to the general concept of executing UML models, not a specific product. In this thesis we are interested in the executable UML described in the Object Management Group (OMG) Submission *Semantics of a Foundational Subset for Executable UML Models* [12]. We refer to it as Executable UML, using upper-case E, and the term includes the infrastructure for working with Executable UML: construction, validation, transformation and execution of an fUML model. The term fUML refers to the "modeling language" (a subset of UML), which is used in Executable UML to define an executable model.

1.2 Background

The evolutionary steps of software development tend to be towards higher abstraction: from machine languages to assembly languages to high-level languages to domain-specific language and so forth. In parallel to these programming languages, there are modeling languages. A commonly known modeling language is the Unified Modeling Language (UML), which is the accepted industry standard for software modeling. UML is standardized by the OMG and consists of several diagram types, of which UML Activities

are the focus of this thesis.

Executable UML is a software modeling concept from OMG that allows execution of a modeled behavior with no or very little human interaction. The idea is that a software engineer can model the design of a system, which is then executable, either directly or through transformation. In Executable UML, Activities are used to describe behavior in a data-flow manner. It allows models to directly or indirectly be used in the same way as code—as an actual part of an implementation.

As the use of Object-Oriented Analysis and Design (OOAD) has increased, the use of structural models such as UML Class Diagrams have become widely used in software development. A software engineer can model and implement the structure of a program at the same time by generating a framework from the model. The framework can then be used for implementing the behavior. OMG is now suggesting means to also integrate the dynamic aspect of programming into the modeling phase, that is to enable a visual programming tool for behavior. By combining already accepted and widely used standards together with a standard for well-defined run-time semantics, the OMG enables the standardization of a tool-chain that makes it possible to model and implement behavior at the same time. The objective of the Executable UML specification is to

> *enable a chain of tools that support the construction, verification, translation, and execution of computationally complete executable models.* [12, p.8]

The models are expressed in the Foundational UML Subset (fUML), a subset of UML with well-defined semantics. fUML is the basis for expressing all behavior in Executable UML and is defined in the Executable UML specification. For the tool-chain to link together, all tools must speak the same language, both in terms of underlying format and semantics. *Construction* means to use a tool to create models and corresponds to using a text editor for code. *Verification* refers to the possibility of static analysis, like the one performed on code by a compiler. A *translation* tool also corresponds to

a compiler in the sense that it performs the mapping between higher-level concepts of UML to executable form (but still a model). In addition to being a compiler in a traditional sense, a translation tool could also be intended to map from executable form to code. Finally, the *execution* of models refers to the interpretation of a model, ultimately resulting in the corresponding execution phenomena on some platform.

1.3 Related Work

fUML falls under the categories visual programming languages and data-flow languages. A similar data-flow language is described by Dennis [8]. fUML is essentially UML Activities, which has a semantics that are similar to Petri-nets. This has been used by Farooq et al. in [9] where they propose a transformation methodology for Activity Diagrams into Colored Petri Nets.

An interpreter (or Virtual Machine) for UML Activities is presented in [7], where a textual syntax is used to define the Activities. In relation to this, the same authors have presented a work on clarifying the semantics and pragmatics of UML Actions in [6].

Other approaches to executable UML has been introduced before. In referenced work, the name Executable UML refers to another kind of executable UML. The modeling software company Kennedy Carter uses a well-defined subset of UML called xUML [14] in their *Model Driven Architecture (MDA) with Executable UML*, with which it is possibly to translate from testable platform-independent models to target-specific code. xUML makes use of the Action Specification Language to achieve this. In the similar xtUML [11], used in Mentor Graphics BridgePoint, it is possible to create executable models for a system, verify the system and then generate code. The behavior is specified in UML State Machines and the Object Action Language. The biggest differences compared to the Executable UML in this thesis, is that both xUML and xtUML requires code generation and use UML State Machines whereas we execute the actual models directly and use UML Activities. Both Kennedy Carter and Mentor Graphics are contributors to the

Executable UML submission that we examine in this thesis.

1.4 Motivating Example

Saab AB produces and markets various kinds of military and civil command- and control systems. One major component of such a system is the Human Machine Interface (HMI), through which the operator can control and observe the various subsystems. For example, it provides tactical information and weapon firing capabilities. The tactical information contains a lot of measurements (e.g., position, speed and bearing) which can be represented in the HMI in various units (e.g., m/s and radians) and in various coordinate systems (e.g., Cartesian and polar).

The idea is to use Executable UML to transform data into the desired representation on the screen—i.e., information in some form flow from the data layer into fUML models which transform the data into something that is presented to the operator.

1.5 Problem Specification

We can identify four linked tasks:

- research and collect information about the proposed standard [12],
- implement the execution engine described in that standard,
- define an Application Programming Interface (API) for interacting with that execution engine, and
- show that models can be executed through that API.

The usability of Executable UML for our application is interesting from two perspectives: in terms of how easy it is to create and work with models; and in terms of performance.

1.6 Delimitations

This thesis concentrates on the *execution* of models, even though the standard is concerned with other aspects of executable models—such as construction, validation and translation. A precondition to executing models is of course that we can construct models, but this is not the main focus. Validation and translation has not been considered at all—we are only interested in directly executable models, i.e, fUML models.

We are aiming at a straight-forward implementation of the execution engine—optimizations are left for future work. The reason for this is that it is more important to get a working implementation to start practicing Executable UML and to gain knowledge about how it can be used.

Chapter 2

Executable UML

2.1 The Specification

The *Semantics of a Foundational Subset for Executable UML Models* [12] is essentially comprised by two parts: one that covers fUML and one that describes the Execution Model. These are covered separately below. The relationship between the Execution Model and fUML is that the Execution model defines an execution engine, that in turn realizes the semantics of the fUML.

At the time of this project, the specification was not yet finalized. The version used here is the initial submission. In addition to the two parts mentioned above, a third part called the *Foundational model library* is announced in the submission. However no work is presented yet, but according to the table of contents, it will deal with primitive data types, primitive functions and input/output.

Before we deal with fUML and the Execution Model, we will explain the two UML concepts that constitute the basis for Executable UML: Actions and Activities.

2.2 UML Actions

Actions are the atomic units of behavior in UML, i.e., what can be expressed. An action can take input and deliver output using object nodes called *input pins* and *output pins*. UML specifies about 40 actions, spanning a wide range of fundamental execution phenomena—for example to retrieve the value of an object's property or to invoke an operation on an object. The actions we have considered are listed in Table 2.1. Actions do not exist by themselves, they are always part of a higher-level formalism for expressing behavior—for example activities or state machines.

The UML Superstructure divides actions into levels of functionality. The *basic level* defines actions as having input and output pins and facilitates communication like operation calls, signal sends and behavior calls. The *intermediate level* includes primitive actions such as object creation, object destruction and reading and writing structural features etc. The *complete level* adds further read/write functionality and supports the acception of events.

For further reading, see Conrad Bock [1] and [2]. The UML Superstructure [13, Ch.11] also summarizes actions in an understandable way, before going into details on each action.

Action	Description	Java correspondence
AddStructuralFeatureValueAction	Adds values for a structural feature.	Dot notation to set attribute value, e.g., object.setFoo(f).
CallBehaviorAction	Invokes a behavior directly, rather than invoking a behavioral feature.	Static method call.
CallOperationAction	Invokes an operation on an object, where it may cause the invocation of associated behavior.	Method call on object.
CreateObjectAction	Creates an object that conforms to a statically specified classifier.	The **new** keyword and the default constructor.
OpaqueAction	An action with implementation-specific semantic.	The body of a method.
ReadSelfAction	Retrieves the host object of an action.	The **this** keyword.
ReadStructuralFeatureAction	Retrieves the values of a structural feature.	Dot notation to get attribute value, e.g., object.getFoo().
ReadVariableAction	Retrieves the value of a statically specified variable.	Read local variable.
TestIdentityAction	Tests if two values are identical objects.	The **equals** method.
ValueSpecificationAction	Evaluates a value specification.	No obvious correspondence.
WriteVariableAction	Modifies the value of a statically specified variable.	Write to local variable.

Table 2.1: Actions

2.3 UML Activities

Activities specify behavior. They can contain actions and coordinate execution flows between these actions. In a comparison to traditional programming, an activity can be thought of as a method or a full program and actions as more primitive programming constructs such as object creation, method calls. In a high-level description of Activities the core concepts are *nodes* and *edges*. Nodes belong to one of three categories: control nodes, object nodes or action nodes. Edges can be either control flows or object flows. The UML Superstructure defines that an activity

> *specifies the coordination of execution of subordinate behaviors, using a control and data flow model.* [13, p.328]

This statement captures the relationship between the concepts mentioned above. More explicitly this can be expressed as:

- Control nodes coordinate flows.
- Flows transfer data (objects) and control between nodes.
- Object nodes temporarily hold data.
- Action nodes (executable nodes) operate on data and perform work.

An activity can communicate with other model objects. It can take input and deliver output through *activity parameter nodes*; it can use actions for invoking operations and behaviors, and it can send and receive signals. This is described in more detail below where we discuss the run-time semantics of activities. Figure 2.1 shows an overview of the graphical elements that can be used in an activity.

As with actions, the functionality is divided into a number of levels: fundamental, basic, intermediate, complete and structured. The fundamental level is the top-level and merely defines activities as having nodes. The basic level introduces control and data flow; and the intermediate level contains concurrency and decisions. The complete level is outside our scope, since it

is not included in the foundational subset—which we will talk about in Section 2.5. The structured level includes traditional programming constructs such as loops, conditionals and exception handling.

For further reading, see Conrad Bock [1], [3] and [4]. For details on activities and their packages, see the UML Superstructure [13, Ch.12].

2.3.1 Run-time Semantics

The run-time semantics of a modeling concept refer to the behavior of that concept if it was executed in an execution environment. The UML Superstructure specifies run-time semantics as

> *a mapping of modeling concepts into corresponding program execution phenomena.* [13, p.10]

It also states two fundamental premises about UML semantics:

- any *behavior* is the direct consequence of an action executed by an *active object*, and
- all behavior is event-driven.

Definitions of the concepts behavior and active object is in place. Behavior describes the changes in an object's state, where the state in turn is described by the object's structural features. An active object is an object that, as a direct consequence of its creation, commences to execute its classifier behavior, and does not cease until either the complete behavior is executed or the object is terminated by some external object.

So far we have talked about run-time semantics in general terms. We will now go into further detail on the specific semantics of activities. We have already established that activities utilize a control and data flow model. More specifically—what actually flows in the model are tokens. Tokens can either carry data in the form of objects, or they can transfer control. Control and object tokens have in common that they carry their information from one node to another, and that they are unique, even if they carry identical information.

Flow and Execution

Imagine two nodes connected by a control flow. The second node receives control when the first node offers a control token on the control flow connecting the nodes. That does not necessarily mean that the second node can start executing, since there are a number of conditions that need to be satisfied before execution can begin. What the conditions are depend on the node type. A minimum condition is that every incoming flow to a node offers tokens.

When the preconditions for an action are satisfied, the action takes tokens from all incoming flows and starts its own execution. When execution is done, the action offers tokens on all its outgoing edges and terminates itself.

Concurrency

Activities support concurrent flows. This does not imply the use of parallel computing in the underlying architecture. It is merely a statement saying that the order of execution is not important, the flow can still be "executed" sequentially.

Invocation and I/O

Activities can be invoked in three ways:

- as a *classifier behavior* invoked on instantiation of the classifier,
- as a *method* of an invoked behavioral feature, or
- by direct invocation.

For more information on behavior invocation, see Conrad Bock [2, Sec.4].

2.3.2 Graphical Syntax

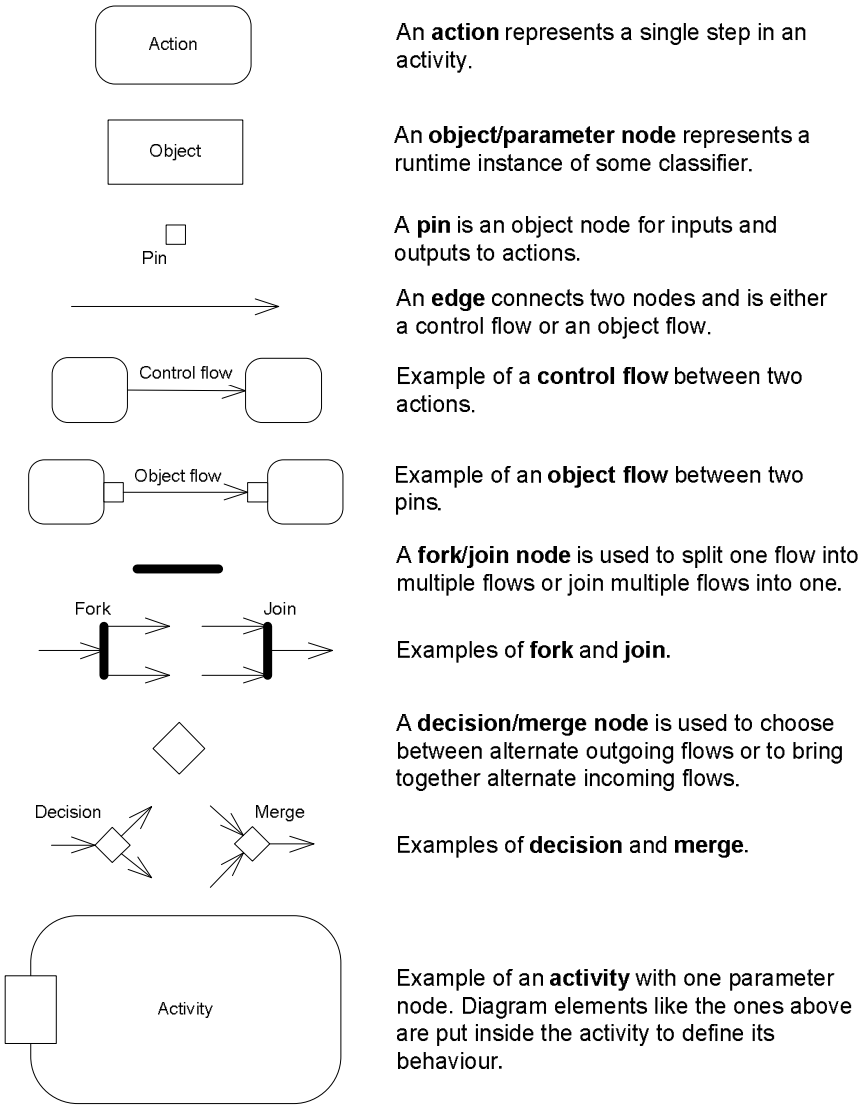

Figure 2.1: Graphical syntax of an activity

2.4 Example Part I: Defining a Model

All the necessary background has been presented. Now we are ready to define an fUML model. We are aiming for something small and simple, that still illustrates the concept—and to relate back to the motivating example, we have chosen to model the Polar-to-Cartesian (P2C) transform.

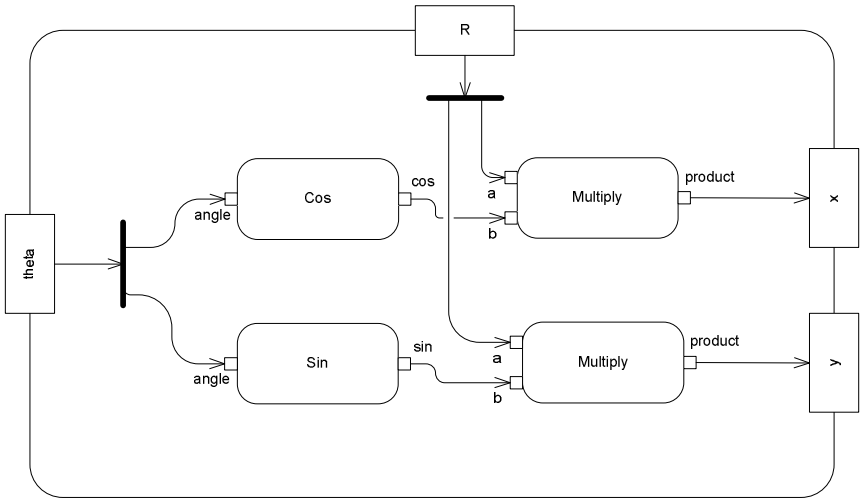

Figure 2.2: The Polar-to-Cartesian activity

The execution flow is as follows:

- The value of **theta** is available in a token on its activity parameter node.
- The token travels through the object flow to a fork node, which results in a duplication of the token.
- One token travels to **Sin** and one to **Cos**—and these opaque actions for calculating sin and cos are triggered directly since all input pins offer tokens.
- The results of **Cos** and **Sin** travel from the output pins to two instances **Multiply**, another opaque action. However, **Multiply** can not execute until all input pins offer tokens.
- Concurrently—the value of **R** ends up in the two **Multiply** actions in an analogous way.
- The **Multiply** actions can now execute since all input pins offer tokens. The resulting tokens flow from the output pins to activity parameter nodes **x** and **y**—and are now available to the caller.

2.5 Foundational Subset

The Executable UML Specification defines the Foundational UML Subset (fUML) as:

> *an executable subset of standard UML that can be used to define, in an operational style, the dynamic and static semantics of modeling languages such as standard UML or its subsets and extensions.* [12, p.8]

Actions are the primary focus of the specification, since fundamental behavior in UML is expressed by actions. But actions need a context—and the specification use activities as context. When choosing the subset, the contributors have removed components without semantic significance and tried to achieve a balance between compactness and ease of translation. Compactness simplifies the implementation of execution engines, but it also complicates the process of translating from UML to fUML.

The goal of the specification is to create a subset—as small as possible—and still be able to automatically translate a standard UML model into an fUML model. The idea is to use this subset to formally define the semantics of higher level UML constructs. The diagram in Figure 2.3 shows the fUML packages.

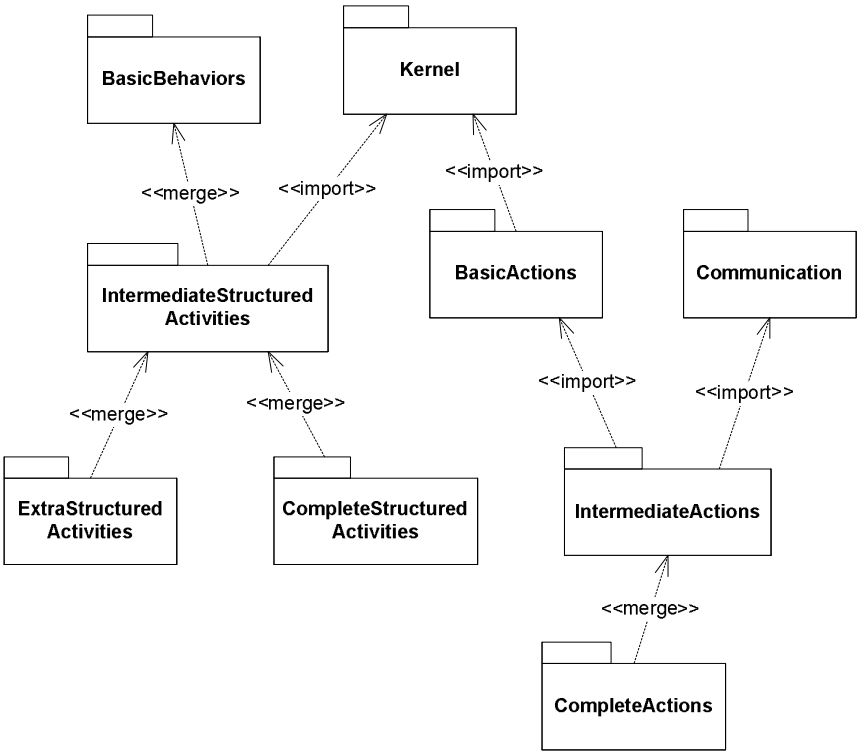

Figure 2.3: The fUML packages

Note that the diagram only shows the fUML subset on package level, i.e., the result of excluding packages from UML that has no members in fUML. Exclusions have been made inside the fUML packages too. For details, see the Abstract Syntax in the Executable UML Specification [12, Ch.7].

2.6 The Execution Model

The Execution Model is designed with inspiration from the *GoF*[1] *Visitor pattern* [10]—a programming construct that enables the separation of algorithms from object structures. In this case it is implemented as an association from an *execution* class to an *executable* class. The relationship is depicted in Figure 2.4.

[1]Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides are often referred to as the Gang of Four or GoF.

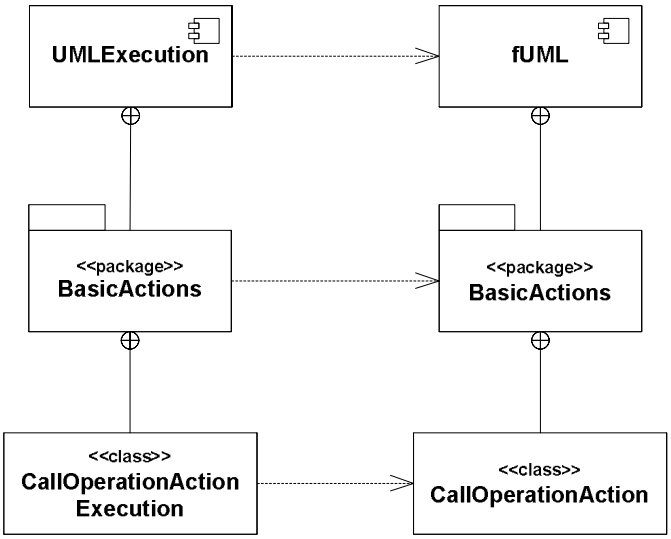

Figure 2.4: Relationship between the Execution Model and fUML

The diagram shows three levels of abstraction. On the top-level there is a UMLExecution component that realizes the semantics of the entire fUML. Then, for each package in fUML, there is a corresponding package that realizes that package's semantics. On the most detailed level, there is a corresponding execution class for each executable fUML class. As we will see later, the execution engine compiles instances of executable classes into instances of the corresponding execution classes.

In addition to a corresponding package for every fUML package, the Execution Model has a package called Locations. It adds no execution semantics, but offers a compact interface for handling scope, compilation and interpretation. It does this through its three classes: Location, Compiler and Interpreter.

An implementation of the Execution Model, i.e., an execution engine, could be considered a virtual machine for fUML models. This justifies the following comparison to Java.

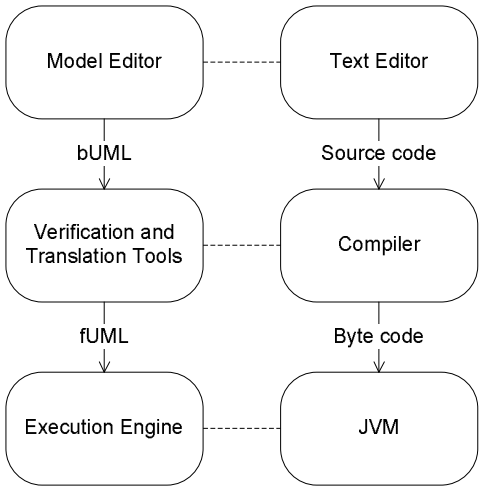

Figure 2.5: Comparison between Executable UML and Java

Chapter 3

Implementation

3.1 Tools

Java

The programming language is Java. It is a natural choice, as the Execution Model is inherently object-oriented and Java is the most wide-spread object-oriented language. Another reason is that we aim for open architectures and Java offers extensive support for various open standards and is itself an open source product.

Eclipse

The development environment is Eclipse. A major reason for this is that Eclipse offers very good Java support. Also, there is a lot of easily accessible functionality in form of pluggable features for Eclipse, e.g., the EMF.

EMF

The Eclipse Modeling Framework (EMF) is a tool for domain-specific modeling. It is a Java framework and code generation facility for building applications from structured (formal) data models. EMF unifies three major technologies in modern software engineering, namely UML, Java and eXtensible Markup Language (XML). Since fUML models are UML models and

the execution engine itself makes references to UML elements, we need an implementation of UML. The obvious choice is the EMF-based implementation of UML 2.1, packaged as an Eclipse feature. Among other advantages, this choice enables integrated code generation in Eclipse.

OSGi

The execution engine is deployed as an Open Services Gateway initiative (OSGi) bundle. An OSGi bundle is essentially a Java Archive (JAR) file containing class files and a manifest. Bundles are installed in the OSGi framework which is running on top of a Java Virtual Machine (JVM). Our bundle includes the implementation of the execution engine and provides an interface for client applications to use.

3.2 Iterative and Model-Driven Development

Since the execution engine is inherently object-oriented and described in UML notation, the method we chose to implement the engine almost presented itself to us and includes the following steps:

- create an EMF (UML2) model of the Execution Model,
- generate a code framework from the EMF model, and
- implement the execution engine's behavior by filling in the blanks in the generated code base.

Naturally, errors were made in our EMF representation of the Execution model and also in the specification itself—especially since it is not yet finalized. Therefore the model representation has been refined in an iterative manner throughout the implementation phase. The process of building up the EMF model forced us to examine the Execution Model in detail, hence gaining knowledge about the system it describes. The implementation method is depicted in Figure 3.1. For details on how to use EMF, see Budinsky [5].

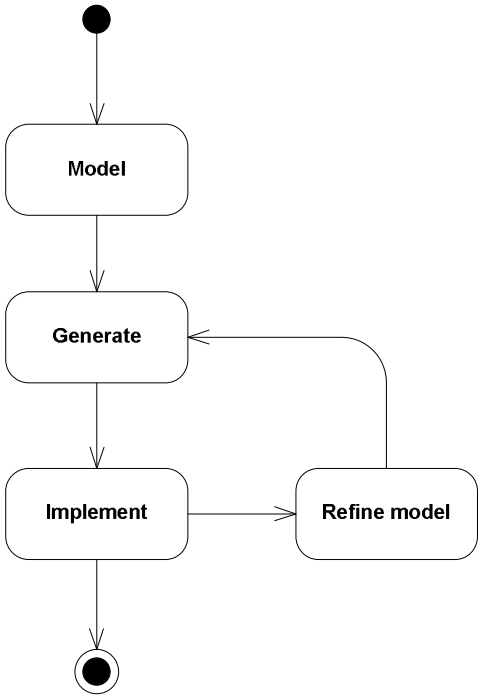

Figure 3.1: The iterative implementation method

Creating the EMF Model

The first thing to do is to create a new *UML2 Model*[1] The top-level model element is given the name **umlexecution**. We are now ready to add the actual model elements from the specification. The procedure is as follows:

- Add the standard UML meta model as a resource to the model. This is needed because references will be made to the UML meta model.
- Add all packages as children to **umlexecution** and set the name for each package.
- Add all classes contained in each package as children to that package and set the name for each class.
- Add all associations contained in each package as children to that package and set multiplicities, member ends etc.

[1]In Eclipse: File → New → Other → UML Model. The model object should be Model.

- Add all generalizations as children to the specializing class and set the generalizing class.
- For each class, add all owned properties and set multiplicities, types etc.
- For each class, add all owned operations and set parameters, types etc.

We now have a complete EMF representation of the Execution Model.

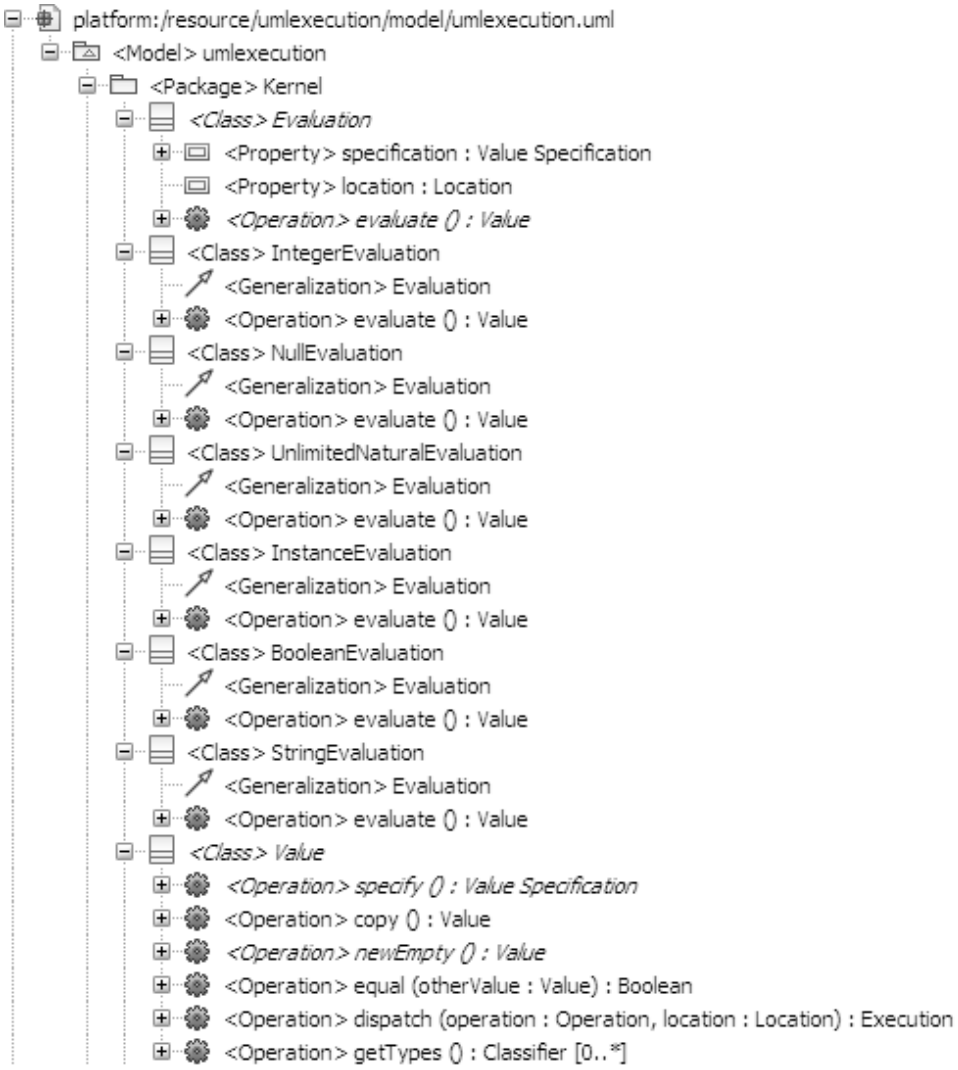

Figure 3.2: A partial view of the Execution Model represented in EMF

Figure 3.2 shows a partial EMF UML2 Class Model, which is our representation of the Execution Model from 2.5. The Class model contains all the Execution Model packages (only Kernel is visible here) with their classes. Each class has its properties and operations specified. This EMF model is used to generate the code framework for implementing the execution engine.

Code Generation and Manual Coding

How to generate the code framework from the EMF model is outside the scope of this thesis. Please refer to the Budinsky et al. [5] to learn about how to generate efficient code with EMF.

The generated code serves as a first draft of an implementation of the execution engine. Now the implementation needs actual functionality, which is achieved by manually implementing the behavior in code. This part of the implementation process requires a more thorough examination of the specification than previous parts. We need to understand the overall concepts and how they interconnect, as well as the details, which are described in the specification in numerous textual descriptions and diagrams. A thorough study of selected parts of the UML Superstructure is also necessary. Chapter 2 summarized all of these concepts.

3.3 Breaking Down the Execution Model

Apart from the Execution Model as a whole, we need to identify and prioritize what specific functionality we want to focus on. This is why we broke down the Execution Model into a number of tasks, which are listed below.

Identified task	Description
Action execution	General support for executing actions.
Active objects	Support for objects to have a classifier behavior.
Activity node execution	General support for executing nodes.
Conditionals and loops	Specific implementation of conditional nodes and loop nodes.
Control nodes (except fork)	Support for all control nodes except fork/join.
Evaluations	General support for evaluating value specifications.
Fork node execution	Support for fork and join of flows.
Invocation actions	Support for call behavior and call operation.
Links	Support for working with associations between objects.
Locations package	Contains the context in which executions are created.
Opaque execution	Implementation specific behavior which can be used to implement primitive operations.
Primitive values	Support for value specifications of primitive values.
Signals	Support for sending/receiving signals between objects.
Structural features	Support for actions that operate on structural features.

Table 3.1: Identified tasks

3.4 Creating and Storing a Model

The construction of fUML models is done in the EMF UML2 editor—a tree-view editor for UML models. In a complete tool-chain this step would be done in a graphical modeling tool that uses the same underlying format as

the rest of the tool-chain. Since we do not have such a tool[2] we will have to do this in two steps: first graphically define the model to show what it will look like in a complete tool-chain; and then construct the model.

Since the syntax of fUML is graphical, we need a format to store the model. The specification does not specify on what format the model is to be stored; it does however imply that the XML Metadata Interchange (XMI) is a good choice. XMI is an OMG standard primarily used for distributing and storing UML models. It is a dialect of XML. Models created in the EMF UML2 editor *are* in fact stored on this format, so if we make the execution engine use the same format, we have a beginning tool-chain.

Figure 3.4 shows the XMI representation of the P2C model from Section 2.2. The format is rather straightforward and can easily be mapped onto the diagram. Most XMI elements have a corresponding graphical element even though there are exceptions, e.g., Parameters, which are invisible elements.

3.5 Example Part II: Creating a Model

The P2C model that we defined in Section 2.2 was just a diagram without underlying model. Now it is time to construct the actual model. Here follows a description on how to create the executable P2C model and a figure showing the full P2C model in the EMF UML2 Editor. For more details on how to work with models in EMF, see Budinsky [5].

1. Create a new **UML Model** with filename **p2c.uml** and model object **Model**.

2. Open the Properties view. (Right-click the Model and select Show Properties View.)

3. Create an **Activity** and give it the name **PolarToCartesian** in the

[2] We tried the Graphical Modeling Framework (GMF) editor, but it is still to immature and was just slowing us down.

properties view. (Right-click the Model and select New Child → Packaged Element → Activity.)

4. Create the **Parameters** with names as below. (Right-click the Activity and select New Child → Owned Parameter → Parameter.)

5. Create the **Activity Parameter Nodes** with names as below. (Right-click the Activity and select New Child → Node → Activity Parameter Node.)

   (a) Add a **Literal Null** for **Upper Bound**[3].

   (b) Connect the corresponding **Parameter** in the properties view.

6. Create the **Fork Nodes** with names as below. (Right-click the Activity and select New Child → Node → Fork Node.)

7. Create the **Call Behavior Action Nodes** with names as below. (Right-click the Activity and select New Child → Node → Call Behavior Action.)

   (a) Add **Input** and **Output Pins** as below and give them **Literal Null** as **Upper Bound**[4].

8. Create the **Object Flows** with names as below. (Right-click the Activity and select New Child → Edge → Object Flow.)

   (a) Connect the **Source** and **Target** as the names suggest below. The naming convention used here is [source]_[target].

   (b) Add a **Literal Null** for **Guard** and **Weight**[5].

9. Save the **p2c.uml** model.

[3]Upper Bound is a required feature on Activity Parameter Node, so this is needed to pass validation.

[4]Upper Bound is a required feature on Pins, so this is needed to pass validation.

[5]Guard and Weight are required features on Object Flows, so this is needed to pass validation.

Try running the validation by right-clicking the Model and select Validate. The result will be "The required feature 'behavior' of '<Call Behavior Action> call' must be set". We need to create and connect the math functions that our P2C model calls. These are implemented as opaque behaviors and will be located in a common library that many models can use. Right now the library works as an interface for other models—the actual implementation is in Java code in the execution engine. A next step is too move the actual implementation into the model object itself.

1. Create a new **UML Model** with filename **math.uml** and model object **Model**.

2. Create the **Function Behaviors** with names as below. (Right-click the Model and select New Child → Packaged Element → Function Behavior.)

   (a) Add **Parameters** as below.

3. Validate and save the **math.uml** model.

4. Go back to **p2c.uml** and load **math.uml** as a resource. (Right-click the model and select Load Resource → Browse Workspace and find math.uml)

5. Go back to the **Call Behavior Action Nodes** and connect the corresponding **Behavior** in the properties view.

6. Save the model and try to validate it again. This time it should work!

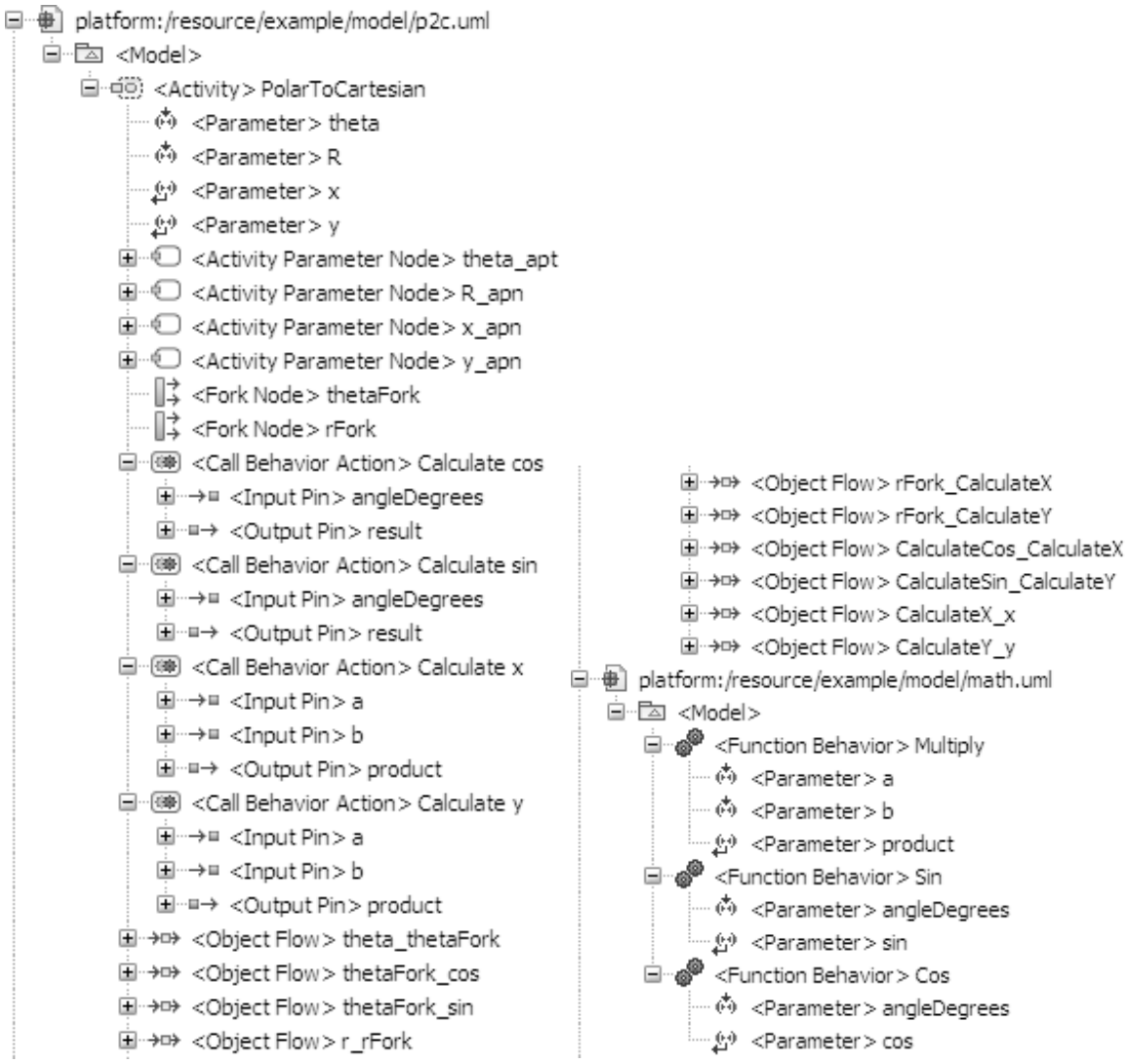

Figure 3.3: EMF representation of the P2C model

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML" xmi:id="_yDXfQChREd-vA56N8Q1_yQ">
  <packagedElement xmi:type="uml:Activity" xmi:id="_7uPCEChREd-vA56N8Q1_yQ" name="PolarToCartesian">
    <ownedParameter xmi:id="_EGqL8ChaEd-vA56N8Q1_yQ" name="theta"/>
    [...]
    <node xmi:type="uml:ActivityParameterNode" xmi:id="_UZsPMChSEd-vA56N8Q1_yQ" name="theta"
          outgoing="_heBoMChSEd-vA56N8Q1_yQ" parameter="_EGqL8ChaEd-vA56N8Q1_yQ">
      <upperBound xmi:type="uml:LiteralInteger" xmi:id="_00cM0CheEd-vA56N8Q1_yQ" value="1"/>
    </node>
    [...]
    <node xmi:type="uml:ForkNode" xmi:id="_aBFVgChSEd-vA56N8Q1_yQ" name="thetaFork"
          outgoing="_rDpeoChSEd-vA56N8Q1_yQ _ACy4cChbEd-vA56N8Q1_yQ" incoming="_heBoMChSEd-vA56N8Q1_yQ"/>
    [...]
    <node xmi:type="uml:CallBehaviorAction" xmi:id="_aKCHEChbEd-vA56N8Q1_yQ" name="Cos">
      <argument xmi:id="_aKCHEihbEd-vA56N8Q1_yQ" name="angle" incoming="_rDpeoChSEd-vA56N8Q1_yQ">
        <upperBound xmi:type="uml:LiteralInteger" xmi:id="_5hMP0CheEd-vA56N8Q1_yQ" value="1"/>
      </argument>
      <result xmi:id="_aKCHEyhbEd-vA56N8Q1_yQ" name="cos" outgoing="_er6E4ChdEd-vA56N8Q1_yQ">
        <upperBound xmi:type="uml:LiteralInteger" xmi:id="_5r4c4CheEd-vA56N8Q1_yQ" value="1"/>
      </result>
      <behavior xmi:type="uml:FunctionBehavior" href="math.uml#_XraTEChcEd-vA56N8Q1_yQ"/>
    </node>
    [...]
    <edge xmi:type="uml:ObjectFlow" xmi:id="_heBoMChSEd-vA56N8Q1_yQ" name="theta_thetaFork"
          source="_UZsPMChSEd-vA56N8Q1_yQ" target="_aBFVgChSEd-vA56N8Q1_yQ">
      <guard xmi:type="uml:LiteralNull" xmi:id="_DKV7YChfEd-vA56N8Q1_yQ"/>
      <weight xmi:type="uml:LiteralNull" xmi:id="_Gkwz0ChfEd-vA56N8Q1_yQ"/>
    </edge>
    [...]
  </packagedElement>
</uml:Model>
```

Figure 3.4: XMI representation of the P2C model

Chapter 4

Results

4.1 Execution Engine

A result of this work is a partial Java implementation of an execution engine for fUML models. The parts that were left unimplemented are listed in Table 4.2. The engine itself conforms to the Execution Model in [12]. During the implementation process we have gained knowledge about Executable UML—knowledge that helps us to draw conclusions about the usability of Executable UML in applications like the one described in the Motivating Example in Section 1.4. Table 4.1 revisits Table 3.1, with check marks on finished tasks.

Task	Completed
Action execution	✓
Active objects	✓
Activity node execution	✓
Conditionals and loops	
Control nodes (except fork)	✓
Evaluations	✓
Fork node execution	✓
Invocation actions	✓
Links	
Locations package	✓
Opaque execution	✓
Primitive values	✓
Signals	
Structural features	✓

Table 4.1: Revisiting the tasks

It was early on realized that it was more important to implement the "framework" contructs, i.e., everything but actions, than specific actions. Consequently, a number of actions where excluded, leaving their corresponding execution class unimplemented. Classes representing actions that are easy-to-grasp or commonly used were chosen first, as well as vital abstract actions from which other actions inherit features. Table 4.2 lists classes that were left unimplemented, or just partially implemented.

Unimplemented class	Comment
ClearAssociationActionExecution	*see Links*
ClearStructuralFeatureActionExectution	No added value in terms of evaluating Executable UML.
ConditionalNodeExecution	*see Conditionals and Loops*
CreateLinkActionExecution	*see Links*
DestroyLinkActionExecution	*see Links*
EnumerationValue	No added value.
ExpansionRegionExecution	Low priority and lack of time.
Link	*see Links*
LoopNodeExecution	*see Conditionals and Loops*
ReadExtentActionExecution	Lack of time.
ReadIsClassifiedObjectActionExecution	Lack of time.
ReadLinkActionExecution	*see Links*
ReclassifyObjectActionExecution	Low priority.
ReduceActionExecution	No obvious use in our application.
RemoveStructuralFeatureActionExecution	Lack of time.
SendSignalActionExecution	Lack of time.
TestIdentityActionExecution	Lack of time.

Table 4.2: Unimplemented or partially implemented classes

Conditionals and Loops

ConditionalNode was discarded because it has no graphical syntax and we find it rather complicated to use. There are also indications that DecisionNode is expected in the final Executable UML specification, which may have effect on ConditionalNode so we chose to wait.

LoopNode was also discarded due to lack of graphical syntax and complicated usage.

Links

Links (or associtations) has no obvious use in our application and were therefore low priority, leaving all the associated classes unimplemented.

4.2 Execution API

The Execution API, implemented in the **UMLExecution** class, is the glue between the execution engine and an application that wants to execute models. The class diagram in Figure 4.1 shows UMLExecution and its closest relationships. The white boxes are direct associations and constitutes the actual API. **Value** comes from the Execution Model and **Activity** from the EMF UML2 implementation. The grey boxes are the most important indirect associations.

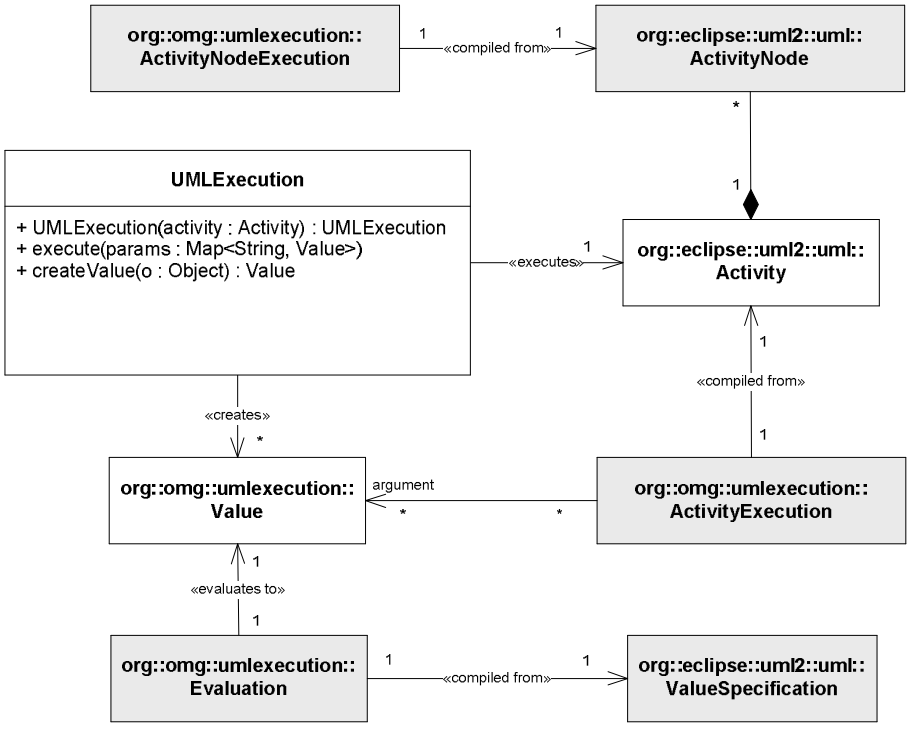

Figure 4.1: Execution API

A user instantiates UMLExecution for an Activity and executes it with

different parameters using the **execute()** method. The parameter map contains name/value pairs for both input and output. Hence, the result from executing the activity can be extracted from the same map. In fact, the activity can change the value for parameters that are both input and output. The Values that goes in the map, can be created using **createValue()**, a convenient method from UMLExecution.

4.3 Example Part III: Executing a Model

Here follows a Java code example for an OSGi BundleActivator that loads the P2C model that we created in Section 3.5 and executes it with the Execution API. The API can of course be used in a pure Java program, without OSGi. The code is commented to describe what the various parts does. The first few lines loads an EMF model in standard EMF manner. Then a UMLExecution instance is created for the loaded model. This instance can be executed multiple times, with different arguments. The next part constructs a map of input arguments—in this case theta=$\pi/6$ and R=1. Then the activity is executed to transform from polar to Cartesian coordinates. The last part prints the results. The expected result is x=0.87 and y=0.5.

```
public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {

        // Load the model from file.
        ResourceSet resourceSet = new ResourceSetImpl();
        URL url = context.getBundle().getResource("model/p2c.uml");
        Resource resource = resourceSet.getResource(URI
                .createURI("bundleresource://"
                        + context.getBundle().getBundleId() + "/"
                        + url.getPath()), true);

        // Extract the model object, which is always the top-level element.
        Model model = (Model) resource.getContents().get(0);

        // Extract the PolarToCartesian activity object from the model.
        Activity activity = (Activity) model.getOwnedType("PolarToCartesian");

        // Create an execution for the activity.
        UMLExecution umlExecution = new UMLExecution(activity);

        // Set and print the input values.
        System.out.println("***** Set input values *****");
        Map<String, Value> paramValues = new HashMap<String, Value>();
        paramValues.put("theta", umlExecution.createValue(Math.PI / 6));
        paramValues.put("R", umlExecution.createValue(1));
        for (String name : paramValues.keySet()) {
            System.out.println(name + "=" + paramValues.get(name));
        }

        // Execute the activity.
        System.out.println("***** Execute activity *****");
        umlExecution.execute(paramValues);

        // Look at the results.
        System.out.println("***** Result *****");
        for (String name : paramValues.keySet()) {
            System.out.println(name + "=" + paramValues.get(name));
        }
        System.out.println("***** Converting back to primitive types *****");
        double x = Double.parseDouble(paramValues.get("x").specify()
                .stringValue());
        double y = Double.parseDouble(paramValues.get("y").specify()
                .stringValue());
        System.out.println("x=" + x);
        System.out.println("y=" + y);
    }

    public void stop(BundleContext context) throws Exception {

    }

}
```

Here follows the result from running the example program. In the first part (Set input values), we use the UMLExecution API to create Values. The first two lines are printed by the execution engine when it compiles the ValueSpecifications to Evaluations. The Evaluations are evalutated to Values, which are then stored in the parameter map. The next two lines show the contents of the parameter map.

The second part (Execute activity) contains the output from the execution engine when it compiles and executes the Activity. The first two lines

come from the actual calls to first compile the Activity and then execute it. The rest follows due to the Just-In-Time (JIT) compiling behavior of the execution engine. When an ActivityNode is ready to execute, it will first be compiled and then executed. If the ActivityNode were to be reused, it would be executed directly, without compiling.

The third part (Result) prints the parameter map again. In addition to the unchanged input parameters, we now have the resulting x and y. Due to the fact that UML does not include a real number primitive type, the results are strings. In the last part these are converted back to regular Java doubles and printed.

```
***** Set input values *****
Compiling ValueSpecification: LiteralString (name: null) -> StringEvaluation
Compiling ValueSpecification: LiteralInteger (name: null) -> IntegerEvaluation
theta=org.omg.umlexecution.impl.StringValueImpl@1f8bd0d (value: 0.5235987755982988)
R=org.omg.umlexecution.impl.IntegerValueImpl@143bf3d (value: 1)
***** Execute activity *****
Compiling Behaviour: Activity (name: PolarToCartesian) -> ActivityExecution
Executing: ActivityExecution
Compiling ActivityNode: ActivityParameterNode (name: theta) -> ActivityParameterNodeExecution
Compiling ActivityNode: ActivityParameterNode (name: R) -> ActivityParameterNodeExecution
Compiling ActivityNode: ActivityParameterNode (name: x) -> ActivityParameterNodeExecution
Compiling ActivityNode: ActivityParameterNode (name: y) -> ActivityParameterNodeExecution
Compiling ActivityNode: ForkNode (name: thetaFork) -> ForkNodeExecution
Compiling ActivityNode: ForkNode (name: rFork) -> ForkNodeExecution
Compiling ActivityNode: CallBehaviorAction (name: Cos) -> CallBehaviorActionExecution
Compiling ActivityNode: InputPin (name: angle) -> InputPinExecution
Compiling ActivityNode: OutputPin (name: cos) -> OutputPinExecution
Compiling ActivityNode: CallBehaviorAction (name: Sin) -> CallBehaviorActionExecution
Compiling ActivityNode: InputPin (name: angle) -> InputPinExecution
Compiling ActivityNode: OutputPin (name: sin) -> OutputPinExecution
Compiling ActivityNode: CallBehaviorAction (name: Multiply 1) -> CallBehaviorActionExecution
Compiling ActivityNode: InputPin (name: a) -> InputPinExecution
Compiling ActivityNode: InputPin (name: b) -> InputPinExecution
Compiling ActivityNode: OutputPin (name: product) -> OutputPinExecution
Compiling ActivityNode: CallBehaviorAction (name: Multiply 2) -> CallBehaviorActionExecution
Compiling ActivityNode: InputPin (name: a) -> InputPinExecution
Compiling ActivityNode: InputPin (name: b) -> InputPinExecution
Compiling ActivityNode: OutputPin (name: product) -> OutputPinExecution
Compiling Behaviour: FunctionBehavior (name: Cos) -> FunctionExecution
Executing: FunctionExecution
Compiling ValueSpecification: LiteralString (name: null) -> StringEvaluation
Executing OpaqueBehaviour Cos. (angle=0.5235987755982988) -> (cos=0.8660254037844387)
Compiling Behaviour: FunctionBehavior (name: Multiply) -> FunctionExecution
Executing: FunctionExecution
Compiling ValueSpecification: LiteralString (name: null) -> StringEvaluation
Executing OpaqueBehaviour Multiply. (a=1.0  b=0.8660254037844387) -> (product=0.8660254037844387)
Compiling Behaviour: FunctionBehavior (name: Sin) -> FunctionExecution
Executing: FunctionExecution
Compiling ValueSpecification: LiteralString (name: null) -> StringEvaluation
Executing OpaqueBehaviour Sin (angle=0.5235987755982988) -> (sin=0.49999999999999994)
Compiling Behaviour: FunctionBehavior (name: Multiply) -> FunctionExecution
Executing: FunctionExecution
Compiling ValueSpecification: LiteralString (name: null) -> StringEvaluation
Executing OpaqueBehaviour Multiply. (a=1.0  b=0.4999999999999994) -> (product=0.4999999999999994)
***** Result *****
theta=org.omg.umlexecution.impl.StringValueImpl@1f8bd0d (value: 0.5235987755982988)
R=org.omg.umlexecution.impl.IntegerValueImpl@143bf3d (value: 1)
y=org.omg.umlexecution.impl.StringValueImpl@11daa0e (value: 0.49999999999999994)
x=org.omg.umlexecution.impl.StringValueImpl@879860 (value: 0.8660254037844387)
***** Converting back to primitive types *****
x=0.8660254037844387
y=0.4999999999999994
```

Chapter 5

Conclusion

Tool-Chain

The execution engine is only one tool in a tool-chain for working with executable UML models. For this model-driven method to add something to the development process, we need to complete the tool-chain. For the first tool in the chain—the construction tool—we have used the EMF UML2 Model Editor. A natural next step is to try existing UML editors that support XMI. The Eclipse GMF UML2 Diagram Editor is a good candidate, but it needs to become more stable first.

Today, software developers expect a lot of support from their development environment, e.g., code completion and content assist in Eclipse. In terms of ease-of-use, fUML very much boils down to the model editor, hence placing high demands on ditto. The EMF UML2 Model Editor that we used in this project is not good enough. As soon as the model becomes only slightly complex it becomes hard to manage:

- A drop-down menu is used to associate objects with each other, e.g., when setting the source and target nodes for a flow. As the number of objects grow it becomes difficult to get an overview. A drag-n-drop feature or model completion (cf. code completion) would simplify this.
- A lot of user interaction, in terms of clicking the mouse in different

places, is required to achieve very little.

- The properties view is separated from the model objects. It would be better to be able to modify properties in the actual object, at least common properties like name.

- Rarely used but still required features like upper bounds must always be set. A default value would simplify.

Furthermore, the fact that the standard does not really specify the format on which models should be stored makes it difficult to mix tools from different vendors. For example, a user might want to buy a mature graphical modeling tool to do the actual modeling, but use an open-source execution engine.

fUML

It is cumbersome to express even the simplest programming constructs, such as loops and conditionals. A good tool support could perhaps simplify this, but still the result is not in any way easier to understand than a traditional programming syntax.

The fact that UML (and therefore fUML) misses a number of primitive types—for example real numbers—is of course a major drawback. We have solved this problem by passing around real numbers as strings and converting to and from real numbers when needed for calculations. This brings us to questions about scalability and performance.

Scalability

We haven't really looked at scalability and performance. But just by reasoning about it, one can argue that large models and/or high data rates will probably reveal scalability and performance issues. The first thing that comes to mind is the number of threads in a large "program", i.e., model. However, this is an implementation detail that can be replaced by a sequential approach. This is a good candidate for future work.

Furthermore, the extra CPU time and memory spent on converting back and forth between primitive types and strings is probably not negligible. To sacrifice CPU and memory in favor of higher level abstractions is a common trade-off, but in the case of fUML models, the gain in abstraction is probably not close to the loss in performance. In fact, writing programs directly in fUML does not offer a higher abstraction than programming Java, it is just another syntax.

Depending on the application—scalability and performance may or may not pose a problem. Anyway, Moore's Law will eventually solve this problem.

Model Library

A large model library and a library of opaque behaviors could potentially help solve parts of the performance issues, as well as increase the usability in terms of simplifying the use of fundamental programming constructs.

References

[1] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4):43–53, July-August 2003.

[2] C. Bock. UML 2 Activity and Action Models, Part 2: Actions. *Journal of Object Technology*, 2(5):41–56, September-October 2003.

[3] C. Bock. UML 2 Activity and Action Models, Part 3: Control Nodes. *Journal of Object Technology*, 2(6):7–23, November-December 2003.

[4] C. Bock. UML 2 Activity and Action Models, Part 4: Object Nodes. *Journal of Object Technology*, 3(1):27–41, January-February 2004.

[5] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley, 2004.

[6] M. L. Crane and J. Dingel. Towards a Formal Account of a Foundational Subset for Executable UML Models. In K. Czarnecki, I. Ober, J-M Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 675–689. Springer, 2008.

[7] M. L. Crane and J. Dingel. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 96–110, New York, NY, USA, 2008. ACM.

[8] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[9] U. Farooq, C. P. Lam, and H. Li. Transformation methodology for UML 2.0 activity diagram into colored Petri nets. In *ACST'07: Proceedings of the third conference on IASTED International Conference*, pages 128–133, Anaheim, CA, USA, 2007. ACTA Press.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[12] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models*, initial submission, ad/06-05-02 edition.

[13] Object Management Group. *Unified Modeling Language: Superstructure*, version 2.0, formal/05-07-04 edition.

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