MASTER'S THESIS

Code generation through model transformation

by

ing. L.A.J. Verstraeten

Supervisors:
prof. dr. M.G.J. van den Brand
ir. L.J.P. Engelen
ir. M. Hamilton

Eindhoven, February 2008
Abstract

This thesis describes how UML models can be transformed into code of an object-oriented imperative programming language. The UML models are modeled with classes and activities. The transformation consists of transformation rules on the metamodel level, between the UML metamodel and the metamodel of the target language.

The transformation rules are based on the reconciliation of the differences between the metamodels. The differences are found in the composition of the metamodels and are reconciled in two transformation steps. A third transformation step transforms the elements in the UML model to elements in the model of the target language.

The transformation rules that were defined during this project, have been implemented with a state-of-the-art tool within the Eclipse framework. The tools are specialized for model transformation.
## Contents

1 Introduction

1.1 Research goal .............................................. 1
1.2 Thesis overview ........................................... 2

2 Preliminaries

2.1 Model Driven Engineering ................................. 3
2.2 UML ....................................................... 3
   2.2.1 UML Classes ......................................... 4
   2.2.2 UML Activities ....................................... 5
2.3 POOSL ..................................................... 7
   2.3.1 The Data layer ....................................... 7

3 Mapping of POOSL data classes to UML

3.1 Overview .................................................. 9
3.2 Mapping the structural part ............................... 10
   3.2.1 Concrete mapping ................................... 10
   3.2.2 Data Class Definition ............................... 11
   3.2.3 Data Method Definition ............................. 12
3.3 Mapping the behavior ..................................... 12
   3.3.1 Concrete mapping ................................... 12
   3.3.2 Data Expressions ................................... 14
   3.3.3 Interpretation of the flow graph ..................... 17
3.4 Conclusions .............................................. 19

4 Differences between the UML and POOSL metamodels

4.1 The POOSL metamodel .................................... 21
4.2 Identifying differences ................................... 21
4.3 Difference in hierarchy .................................. 22
   4.3.1 Data dependencies ................................... 23
   4.3.2 Sequential composition .............................. 24
   4.3.3 Conditional expression .............................. 24
   4.3.4 Loop expression ..................................... 25
4.4 Conclusions .............................................. 26

5 Transformation from UML to POOSL

5.1 Overview .................................................. 27
5.2 Transforming the structural part .......................... 28
5.3 Transforming behavior .................................... 29
   5.3.1 Grouping patterns ................................... 29
   5.3.2 Order groups ........................................ 31
   5.3.3 From UML activities to POOSL expressions .......... 34
5.4 Example .................................................. 35
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.1 The class <em>Dictionary</em></td>
<td>35</td>
</tr>
<tr>
<td>5.4.2 Modeling the <em>Dictionary</em> class in UML</td>
<td>35</td>
</tr>
<tr>
<td>5.4.3 Transforming the UML model to a POOSL model</td>
<td>36</td>
</tr>
<tr>
<td>5.5 Implementation</td>
<td>37</td>
</tr>
<tr>
<td>5.5.1 Tools</td>
<td>39</td>
</tr>
<tr>
<td>5.5.2 Development</td>
<td>39</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>41</td>
</tr>
<tr>
<td>7 Further work</td>
<td>43</td>
</tr>
<tr>
<td>8 Related Work</td>
<td>45</td>
</tr>
<tr>
<td>Appendices</td>
<td>48</td>
</tr>
<tr>
<td>A Data Class Definition</td>
<td>49</td>
</tr>
<tr>
<td>A.1 Inheritance</td>
<td>49</td>
</tr>
<tr>
<td>A.2 Instance Variables</td>
<td>50</td>
</tr>
<tr>
<td>A.3 Instance Methods</td>
<td>50</td>
</tr>
<tr>
<td>A.4 Data class example</td>
<td>50</td>
</tr>
<tr>
<td>B Data Method Definition</td>
<td>53</td>
</tr>
<tr>
<td>B.1 Parameters</td>
<td>54</td>
</tr>
<tr>
<td>B.2 Result</td>
<td>54</td>
</tr>
<tr>
<td>B.3 Local Variables</td>
<td>54</td>
</tr>
<tr>
<td>C Data Expressions</td>
<td>57</td>
</tr>
<tr>
<td>C.1 Using OCL expressions in UML</td>
<td>58</td>
</tr>
<tr>
<td>C.2 Global Variable</td>
<td>58</td>
</tr>
<tr>
<td>C.3 Local Variable</td>
<td>59</td>
</tr>
<tr>
<td>C.3.1 Parameter</td>
<td>60</td>
</tr>
<tr>
<td>C.3.2 Locally defined variable</td>
<td>60</td>
</tr>
<tr>
<td>C.3.3 OCL expression</td>
<td>61</td>
</tr>
<tr>
<td>C.4 Literal</td>
<td>62</td>
</tr>
<tr>
<td>C.5 Return</td>
<td>62</td>
</tr>
<tr>
<td>C.6 Object creation</td>
<td>63</td>
</tr>
<tr>
<td>C.7 Self reference</td>
<td>64</td>
</tr>
<tr>
<td>C.8 Data Method Calls</td>
<td>65</td>
</tr>
<tr>
<td>C.9 Assignment to global variable</td>
<td>66</td>
</tr>
<tr>
<td>C.10 Assignment to local variable</td>
<td>67</td>
</tr>
<tr>
<td>C.10.1 Variable</td>
<td>68</td>
</tr>
<tr>
<td>C.10.2 Property</td>
<td>68</td>
</tr>
<tr>
<td>C.11 Sequential composition</td>
<td>68</td>
</tr>
<tr>
<td>C.12 If</td>
<td>69</td>
</tr>
<tr>
<td>C.13 While</td>
<td>71</td>
</tr>
<tr>
<td>D Validation</td>
<td>73</td>
</tr>
<tr>
<td>D.1 The starting UML model</td>
<td>73</td>
</tr>
<tr>
<td>D.2 Step 1</td>
<td>73</td>
</tr>
<tr>
<td>D.3 Step 2</td>
<td>73</td>
</tr>
<tr>
<td>D.4 Step 3</td>
<td>73</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Ideals Project is a research and development project managed by the Embedded Systems Institute. The goal of Ideals is to develop methods and tools to make embedded software better evolvable. In Ideals, researchers and engineers from ASML have worked closely together with researchers of Delft University of Technology, Eindhoven University of Technology, the University of Twente, the Center for Mathematics and Computer Science (CWI), and the Embedded Systems Institute.

During development, communication between engineers takes place at several abstraction levels. At the highest abstraction levels designs of the system are made in the form of documents and drawings. At the lowest abstraction levels the means of communication describe the physical part of a system, like boards, computers and byte code files. In the model driven engineering vision, models replace the higher level communication artifacts, in such a way that the lowest level artifacts can be derived from the models. Hence model driven engineering refers to the systematic use of models as primary engineering artifacts throughout the engineering life cycle.

The performance of a system can be predicted in an early stage. This is done by executable models that can derive protocol software implementations from modeled application logic. The executable models in the Ideals project are expressed in POOSL.

In the engineering process, engineers design the model of a system. It is convenient if this model is made in one language. UML, as a multiple purpose modeling language, seems to be a valid option. During the engineering process, it is useful to verify different aspects of the model with different paradigms, like POOSL. The UML model cannot be verified directly by one of these paradigms. By transforming the UML model to other languages, the model can be verified during the engineering process.

1.1 Research goal

The goal of this research project is transforming the UML model of a system, using UML activities for behavior, to source code in imperative object-oriented programming languages. We consider the transformation from UML to POOSL data classes as a specific case. A UML model has to be language neutral, meaning that the model is not only made to represent a POOSL program.

The implementation has to work in an Eclipse environment. This constraint is given, so that the current UML tools at ASML and the implementation are all part of one development environment.

Another constraint is that the transformation has to be a model-to-model transformation. This means in our case that a UML model has to be transformed to a POOSL model. These models can be used by different tools, as long as the metamodel is known by both tools.
1.2 Thesis overview

In Chapter 2 model driven engineering is explained in the context of the project. The two languages involved in the process, namely UML and POOSL, are also explained in that chapter.

Chapter 3 shows a mapping from POOSL to UML. The differences of the POOSL and UML metamodels are identified in Chapter 4. Chapter 5 shows the actual transformation from UML to POOSL data classes.

The conclusions of the project are given in Chapter 6.
Chapter 2

Preliminaries

Before defining our transformation, the concepts that play an important role in this project have to be explained. First, we introduce model driven engineering. Furthermore, both UML and POOSL are described in this chapter with the focus on the parts that are relevant for this thesis.

2.1 Model Driven Engineering

Model Driven Engineering (MDE) is a (software) development approach in which the products of every phase of the development life cycle are models. To go from one phase to another in MDE, the model of the first phase has to be transformed into a model of the next phase. This process can be automated, by using a set of transformation rules. Bézivin[3] says the following about models:

"Models are commonly used to provide representation of real-world situations. A model is said to represent a system. [...] Each model is defined in conformance to a metamodel. Metamodels define languages enabling to express models. A metamodel describes the various kinds of contained model elements, and the way they are arranged, related and constrained. A model is said to conform to its metamodel."

An example of a conformance relation between a model and a metamodel is shown in Figure 2.1. The model in this figure shows a table of an administrative system of employees. This table is an instance of the metamodel. The metamodel in this case describes that a table consists of columns that contain values of a specific type. The metamodel is represented as a UML class diagram, which will be described in Section 2.2.1.

Like a model, a metamodel is composed of elements. In a metamodel these are called metaelements. Each element in a model is an instance of a metaelement in the corresponding metamodel. In the example, Employee is an instance of the metaelement Table, EmployeeID and LastName are instances of the metaelement Column, and Int and String are instances of the metaelement Type.

A metamodel is defined in conformance to a metametamodel. The relation between a metamodel and its metametamodel is similar to the relation between a model and its metamodel. These three levels of models are explained by Bézivin[2].

This project is about model transformation, where we transform a UML model into a POOSL model. The transformation is defined on the metamodel level.

2.2 UML

The Unified Modeling Language (UML)[21] is a specification language for object oriented modeling. It is used to create models of a system in a graphical notation.
The syntax of UML is defined by the UML metamodel[11]. The Object Constraint Language (OCL)[20] is used to define constraints on this metamodel, an example of a constraint is that a UML element can never own itself. The semantics of UML is not formally defined, but mostly in informal text documents.

The UML metamodel consists of concrete metaelements. The metaelements in the UML metamodel consist of metaclasses, relations and constraints. An instance of the metamodel is called a repository model[6]. A UML model is an instance of the UML metamodel. This relationship between UML models and the UML metamodel is comparable to the example of Section 2.1. UML models are constructed with UML diagrams in a graphical way. A UML diagram is a partial graphical representation of the modeled system.

In this project two parts of UML are used. We use classes to describe entities in the system, and activities to describe the behavior of the system. UML classes are explained in Section 2.2.1 and UML activities are explained in Section 2.2.2.

### 2.2.1 UML Classes

UML classes describe the structure of the system with entities, their attributes and the relations between classes.

Figure 2.2 shows an example of UML classes. Every class is shown as a rectangle, divided in three parts. The top part shows the class name, the middle part the attributes of the class and the bottom part shows the operations of the class.
2.2. UML

Figure 2.3: Notation of activity nodes

The classes in Figure 2.2 model different shapes. Every shape has a method named `calculateArea`, which calculates the surface area of the shape. There are two kinds of shapes, namely `Rectangle` and `Circle`. The generalization relationship denotes class inheritance and tells us that both these classes are a `Shape`. Both `Rectangle` and `Circle` inherit the attributes and methods of the class `Shape`.

2.2.2 UML Activities

UML activities describe the behavior of (elements of) a system. UML activities specify the sequence and conditions for coordinating primitive behavior.

This section is split up in two parts. The first part is about the concepts of UML activities and the second part describes how to interpret actions.

Activity concepts

Bock[4] describes that UML activities contain nodes and edges to form a flow graph describing the behavior of a system. Control tokens and data values flow through the graph.

The nodes in a UML activity are called activity nodes. There are three kind of activity nodes:

- **Action nodes** operate on control tokens and data values that they receive and provide control and data to other actions;
- **Control nodes** route control tokens and data values through the graph;
- **Object nodes** temporarily hold data values until they are requested by other nodes.

Figure 2.3 shows the notation for some of the activity nodes that are relevant for this project.

The edges in a UML activity are called activity edges. Activity edges are always directed. Figure 2.4 shows the notation of activity edges. Activity nodes are connected by two kinds of activity edges:

- **Control flows** connect actions to indicate that the action at the target end of the edge (denoted by an arrowhead) cannot start until the source action finishes;
- **Object flows** connect object nodes to provide inputs to actions.

Control flows connect actions directly, whereas object flows connect input and output pins of actions, they are represented by little rectangles. The pins with outgoing object flows are called output pins and pins with incoming object flows are called input pins.
CHAPTER 2. PRELIMINARIES

Figure 2.4: Notation of activity edges

Figure 2.5: The UML activity of the method calculateArea

**Actions**

Bock[5] describes actions in the following way:

"Actions are the only elements in UML that can query objects, have a persistent effect on them, invoke operations on them, and invoke behaviors directly. For this reason, actions are sometimes called the "primitive" dynamic elements in UML, since all behaviors must eventually reduce to actions to have any effect on objects, or even to invoke other behaviors."

An example of an action is the action `CreateObjectAction`, which creates an instance of a specified class. This instance is offered to the output pin named `result` as a data value.

Object flows can connect two activity nodes, as shown in Figure 2.4. This is done indirectly, because object flows connect input pins with output pins of actions. It depends on the kind of action how many input and output pins there are. The action `ReadSelfAction` in Figure 2.5, for instance, has no input pins and only one output pin.

An action begins executing when its incoming control tokens and data values are available. An action without incoming control or object flows can start executing immediately.

Figure 2.5 shows an example of a UML activity. It is explained in detail in Section 3.3.1.
2.3 POOSL

The Parallel Object-Oriented Specification Language (POOSL)[13] is a system-level modeling language. A system-level modeling language is used to define executable models of a system that determine the feasibility of the design. POOSL has well-defined formal semantics, which are defined with mathematical axioms and rules. This makes POOSL suited to formally verify and analyze the properties of systems that work with both hardware and software. Models made in POOSL can be tested in SHESim[16], which is a simulation tool for POOSL.

There are three layers in POOSL to develop a model, namely:

- **Architecture layer**, which describes the composition of subsystems;
- **Process layer**, which describes the behavior of processes;
- **Data layer**, which describes the data classes.

The architecture layer describes the system specification of a model, which consists of subsystems and channels. The channels allow the process objects in a subsystem to transmit and receive data from other processes connected to the same channel. This data is described in the data layer.

Only the data layer of POOSL is relevant for this project and is explained in the following section.

### 2.3.1 The Data layer

The data layer consists of data classes. The syntax of the data classes resembles the syntax of other object-oriented imperative programming languages.

In the following example, the class *Rectangle* of Figure 2.2 is implemented as a POOSL data class.

```plaintext
data class Rectangle
  extends Shape

instance variables
  height, width : Integer

instance methods
  calculateArea () : Integer
    | surfaceArea: Integer |
    surfaceArea := height * width;
  return surfaceArea.
```

The name of the class is *Rectangle* and it extends the *Shape* class. The instance variables are *height* and *width*, which are both *Integers*. There is one instance method, namely *calculateArea*.

The method *calculateArea* has no parameters and returns an *Integer*. The only local variable of the method is *surfaceArea* of type *Integer*.

The implementation of the method *calculateArea* consists of two data expressions. The first expression is an assignment of the product of *height* and *width* to *surfaceArea*. The second expression returns the value of *surfaceArea*.

This implementation describes the behavior of the data class. The corresponding UML activity for this behavior is given in Figure 2.5.
Chapter 3

Mapping of POOSL data classes to UML

This project is about the transformation from UML models to POOSL data classes. This is not a trivial step, because the two languages have a different syntax and semantics, as was described in Chapter 2. In this chapter, a mapping is made from POOSL data classes to UML. Although this is the opposite of the desired transformation, this mapping will reveal information that is necessary to realize the desired transformation.

We will use the term aspect to denote the basic concepts of POOSL data classes. An example of such an aspect is class inheritance. Each aspect is mapped to a UML pattern, which is a collection of instances of UML metaclasses.

The mapping yields the following information:

- It indicates whether or not it is possible to transform a UML model to a POOSL model;
- It identifies the patterns in UML that have to be matched.

After the description of the mapping, we give some concluding remarks about our results.

3.1 Overview

This chapter describes the mapping from POOSL to UML. We take data class aspects from POOSL and translate these to UML patterns. Figure 3.1 shows that POOSL code conforms to the POOSL syntax, and that UML models conform to the UML metamodel. The meta level mapping is the mapping of elements of the POOSL syntax to elements of the UML metamodel. The concrete mapping is the mapping of POOSL code to a UML model.

![Figure 3.1: Overview of the mapping from POOSL to UML](image-url)
To structure the mapping, we partition the POOSL syntax into three levels, as it is defined in the thesis of Van Bokhoven[7]:

1. The data class definition;
2. The data method definition;
3. The data expressions.

The first and second level of the syntax describe the structural part of a class, and the third level describes behavior. The following sections describe the structural part and behavior separately, and give a concrete mapping and a meta level mapping for the associated syntax levels.

### 3.2 Mapping the structural part

The structural part of a POOSL data class describes the class interface and method interfaces. This section starts with a concrete mapping. Afterwards the meta level mapping is given for the first two POOSL syntax levels, namely the data class definition level and the data method definition level.

#### 3.2.1 Concrete mapping

In this example we regard the *Rectangle* class, which is defined as follows:

```poosl
data class Rectangle
extends Shape
instance variables
    height, width : Integer
instance methods
    calculateArea (): Integer
    | surfaceArea: Integer |
    | surfaceArea := height * width; |
    return surfaceArea.

setHeight (newHeight: Integer): Integer
    | |
    if newHeight > 0 then
    | height := newHeight
    fi;
    return height.
```

In the rest of this section we describe only the class interface and the method interfaces. The implementation of the methods is described in Section 3.3.

Figure 3.2 shows the *Rectangle* class using UML classes. The class *Shape* has to be modeled to show the inheritance relation.
3.2. MAPPING THE STRUCTURAL PART

![Figure 3.2: The Rectangle class as a UML class](image1)

![Figure 3.3: Subset of the UML metamodel](image2)

### 3.2.2 Data Class Definition

The first level of the POOSL syntax describes the class definition. The syntax of the data class definition is described in a Backus-Naur Form (BNF) like notation:

\[
CD = \text{data class } C \\
\quad [ \text{ extends } C_{\text{super} } ] \\
\quad \text{instance variables } x_1 ... x_n \\
\quad \text{instance methods } MD_1 ... MD_k.
\]

where \(CD\) is the data class definition, \(C\) the class name, \(C_{\text{super}}\) the class to inherit from, \(x_i\) an instance variable and \(MD_i\) a data method definition.

The meta level mapping maps every POOSL data class aspect to a UML pattern. In Figure 3.3, the subset of the UML metamodel that describes the structural part of a UML class is shown. In Table 3.1, all aspects of the POOSL data class definition are listed with their notation in the POOSL syntax and a reference to the metaclass that represents the aspect in UML. This reference consists of a class name, followed by a property of that specific class. A property of a metaclass can either be a value, for instance a string, or an association to another metaclass. A more detailed mapping of the data class definition is given in Appendix A.
Table 3.1: Table of data class definition

<table>
<thead>
<tr>
<th>Poosl aspect</th>
<th>POOSL representation</th>
<th>UML pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class name</td>
<td>C</td>
<td>Class.name</td>
</tr>
<tr>
<td>Class inheritance</td>
<td>C&lt;sub&gt;super&lt;/sub&gt;</td>
<td>Class.superClass</td>
</tr>
<tr>
<td>Instance variables</td>
<td>x&lt;sub&gt;1&lt;/sub&gt; ... x&lt;sub&gt;n&lt;/sub&gt;</td>
<td>Class.ownedAttribute</td>
</tr>
<tr>
<td>Instance methods</td>
<td>MD&lt;sub&gt;1&lt;/sub&gt; ... MD&lt;sub&gt;k&lt;/sub&gt;</td>
<td>Class.ownedOperation</td>
</tr>
</tbody>
</table>

Table 3.2: Table of data method definition

<table>
<thead>
<tr>
<th>Poosl aspect</th>
<th>POOSL representation</th>
<th>UML pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name</td>
<td>m</td>
<td>Operation.name</td>
</tr>
<tr>
<td>Parameters</td>
<td>(u&lt;sub&gt;1&lt;/sub&gt;, ... , u&lt;sub&gt;n&lt;/sub&gt;)</td>
<td>Operation.ownedParameter</td>
</tr>
<tr>
<td>Return type</td>
<td>R</td>
<td>Operation.type</td>
</tr>
<tr>
<td>Local variables</td>
<td></td>
<td>Activity.variable</td>
</tr>
</tbody>
</table>

3.2.3 Data Method Definition

The second level of the POOSL syntax describes the method definition. The syntax of the data method definition is described as:

\[
MD = m(u<sub>1</sub>, ... , u<sub>n</sub>) R
| z<sub>1</sub> ... z<sub>m</sub> |
E
| m(u<sub>1</sub>, ... , u<sub>n</sub>) R |
| z<sub>1</sub> ... z<sub>m</sub> |
primitive.
\]

where MD is the data method, m the method name, z<sub>i</sub> a local variable, u<sub>i</sub> a parameter, R a result type of the data method and E is a data expression.

The data expression is explained in the mapping of behavior in Section 3.3. The keyword primitive is reserved for methods with predefined behavior in POOSL.

The meta level mapping maps every POOSL data class aspect to a UML pattern. In Table 3.2, all the aspects of the POOSL data method definition are listed with their notation in the POOSL syntax and a reference to the metaclass that represents the aspect in UML.

A more detailed mapping of the data method definition is given in Appendix B.

3.3 Mapping the behavior

The implementation of a POOSL data class describes the behavior of data methods with data expressions. Afterwards, a meta level mapping is given for the third POOSL syntax level, namely data expressions. The behavior of data methods in POOSL is mapped to UML activities.

3.3.1 Concrete mapping

As was done for the concrete mapping of the structural part, we regard the Rectangle class, that is defined as follows:
data class Rectangle
extends Shape
instance variables
    height, width : Integer
instance methods
    calculateArea () : Integer
        | surfaceArea: Integer |
        surfaceArea := height * width;
        return surfaceArea.

    setHeight (newHeight: Integer): Integer
        ||
        if newHeight > 0 then
            height := newHeight
        fi;
        return height.

For this concrete mapping of behavior we will look at the behavior of the data methods calculateArea and setHeight.

Figure 3.4 shows a UML activity corresponding to the method calculateArea. The implementation of the method consists of two expressions.

The first expression is an assignment to the local variable surfaceArea. In Figure 3.4, the action labeled &lt;AddVariableValueAction&gt;, with surfaceArea as its variable, represents the assignment of the value of height * length to the variable surfaceArea. The action labeled &lt;CallOperationAction&gt;, with the operation * (multiplication) as its property, is defined to be performed on the integer variable height with the integer variable width as input. The actions labeled &lt;ReadStructuralFeatureAction&gt; read the instance variables of the class. They require the class which they are part of as input (the action labeled &lt;ReadSelfAction&gt;) and have the value of the instance variable as output.

Figure 3.4: The UML activity of the method calculateArea
The second expression in the method `calculateArea` returns the value `surfaceArea`. The node labeled `≪ActivityParameterNode≫` is an object node that represents a value leaving the method, because it has an incoming object flow. The action labeled `≪ReadVariableAction≫` with `surfaceArea` as its variable, reads the local variable `surfaceArea`.

The two expressions are separated by sequential composition, enforcing an explicit order. This order is modeled as a control flow in UML activities.

Figure 3.5 shows a UML activity of the method `setHeight`. This method also consist of two expressions separated by sequential composition.

The first expression of the method `setHeight` is an if statement. In UML activities, the if statement is modeled with two diamond shaped nodes. These nodes are a decision node and a merge node, that are both control nodes. Control nodes can only start executing after receiving a control token. In this example the control token is given by an initial node (represented by a black dot). The decision node has two outgoing flows. Through a path of control flows, a control token eventually arrives in the merge node. These branches represent the choices that can be made. A guard is put on one of the control flows leaving the decision node. The guard is "newHeight > 0", which is also the condition of the if statement in the POOSL code. The guard in the activity is in plain text, since it cannot be modeled otherwise. The flow that has the guard flows into the action labeled `≪AddStructuralFeatureValueAction≫`, which represents the assignment to an instance variable. The action requires the class the instance variable is part of (the action labeled `≪ReadSelfAction≫`), and the value that has to be assigned to the instance variable. The value that is assigned to the instance variable is the parameter `newHeight`. The parameter is modeled with the node labeled `≪ActivityParameterNode≫` with `newHeight` as parameter. The action node finishes the branch of the if statement by having a control flow flowing to the merge node.

The second expression of the method `setHeight` returns the instance variable `height`. This is similar to the return expression of the method `calculateArea`. The incoming control flow comes from the merge node of the if statement.

In this concrete mapping we took two examples describing different behavior. We took the liberty of restraining several constructions (like having only two flows out of the decision node) to map the POOSL data expressions to UML. The meta level mapping is explained in the next section.

### 3.3.2 Data Expressions

The third level of the POOSL syntax describes the data expressions. The syntax of the data expression is described as:
3.3. MAPPING THE BEHAVIOR

\[ E = x \quad \text{global variable} \]
\[ | u \quad \text{local variable} \]
\[ | \gamma \quad \text{literal} \]
\[ | \text{return } E \quad \text{return} \]
\[ | \text{new}(C) \quad \text{object creation} \]
\[ | \text{self} \quad \text{self} \]
\[ | E \, m(E_1, ..., E_n) \quad \text{dynamic method call} \]
\[ | E \, m_C(E_1, ..., E_n) \quad \text{static method call} \]
\[ | x := E \quad \text{assignment to global variable} \]
\[ | u := E \quad \text{assignment to local variable} \]
\[ | E_1; E_2 \quad \text{sequential composition} \]
\[ | \text{if } E_c \text{ then } E_1 \text{ else } E_2 \text{ fi conditional expression} \]
\[ | \text{while } E_c \text{ do } E \text{ od loop expression} \]

The sequential composition, the conditional expression and the loop expression set the execution order of expressions in a method. We call these three expressions *control expressions*. The mapping of the data expressions to UML patterns is described in the next two sections. The first section describes the mapping of the control expressions to UML patterns, and the mapping of the other expression is described in the second section.

**Control expressions**

Control expressions define the order in which expressions have to be executed. For each different kind of control expression we give a mapping.

The sequential composition has the following representation in POOSL:

\[ E_1; E_2 \]

The expression means that \( E_1 \) is executed first and \( E_2 \) second.

In UML activities the sequential composition is represented by connecting an action node of expression \( E_1 \) with an action node of expression \( E_2 \) with a control flow.

The sequential composition mapped to UML does not have a return value, unlike in POOSL expressions. In POOSL expressions, the sequential composition has the second expression (\( E_2 \)) as return value.

The conditional expression in POOSL has the following two representations:

\[ \text{if } E_c \text{ then } E_1 \text{ else } E_2 \text{ fi} \]
\[ \text{if } E_c \text{ then } E_1 \text{ fi} \]

The expression \( E_c \) is the condition. Expression \( E_1 \) and \( E_2 \) are two branches consisting of expressions, where only one option can be chosen. When the condition \( E_c \) holds, the branch of \( E_1 \) is chosen, otherwise the branch of \( E_2 \).

The conditional expression in the UML mapping is represented by a condition node with two outgoing control flows, where one flow has a guard, namely the condition. The two paths eventually join in a merge node. An example of the conditional expression in UML activities is given in Figure 3.5.

The conditional expression mapped to UML does not have a return value, unlike in POOSL expressions. In POOSL expressions, the conditional expression has the chosen branch as return value.

The loop expression in POOSL has the following representation:

\[ \text{while } E_c \text{ do } E \text{ od} \]
The expression $E_c$ is the condition. Expression $E$ is a nonterminal, that represents the body of the loop expression.

The loop expression in UML activities can be modeled in several ways. In this mapping, only the loop node is considered. The loop node is an action node, which contains an expression, for modeling a condition, and nodes and edges, which represent the body of the loop expression.

Figure 3.6 shows an example of the loop node in UML activities. The usage of the loop node is not defined fully in UML. The UML activity in Figure 3.6 represent the behavior:

$$\text{while } a < b \text{ do } a := a + a \text{ od}$$

A more detailed mapping is given in Appendix C.

### Other expressions

The expressions, that are not control expressions, are mapped to activity nodes in UML activities. Table 3.3 shows the UML activity nodes that correspond with the expressions. For each expression, we give a short description.

<table>
<thead>
<tr>
<th>POOSL aspect</th>
<th>POOSL representation</th>
<th>UML pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global variable</td>
<td>$x$</td>
<td>AddStructuralFeatureValueAction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ReadSelfAction</td>
</tr>
<tr>
<td>Local variable (parameter)</td>
<td>$u$</td>
<td>ActivityParameterNode</td>
</tr>
<tr>
<td>Local variable (locally defined)</td>
<td>$u$</td>
<td>ReadVariableAction</td>
</tr>
<tr>
<td>Literal</td>
<td>$\gamma$</td>
<td>ValueSpecificationAction</td>
</tr>
<tr>
<td>Return</td>
<td>return $E$</td>
<td>ActivityParameterNode</td>
</tr>
<tr>
<td>Object creation</td>
<td>new$(C)$</td>
<td>CreateObjectAction</td>
</tr>
<tr>
<td>Self</td>
<td>self</td>
<td>ReadSelfAction</td>
</tr>
<tr>
<td>Assignment to global variable</td>
<td>$x := E$</td>
<td>AddStructuralFeatureValueAction</td>
</tr>
<tr>
<td>Assignment to local variable</td>
<td>$u := E$</td>
<td>AddVariableValueAction</td>
</tr>
<tr>
<td>Method call</td>
<td>$E \ m(E_1, \ldots, E_n)$</td>
<td>CallOperationAction</td>
</tr>
</tbody>
</table>

Table 3.3: Mapping of other expressions to UML patterns
3.3. MAPPING THE BEHAVIOR

Reading global variables is mapped to a \textit{ReadStructuralFeatureAction} node in UML activities. This node requires the global variable to read, and the instance of the class which holds the global variable. The latter is modeled with the \textit{ReadSelfAction}. An example of this UML pattern is shown in Figure 3.4, where \textit{length} and \textit{width} are the global variables that have to be read.

A local variable is the value of either a parameter of a method or a variable that is declared locally. The value of a parameter can be read with the object node \textit{ActivityParameterNode}. This node requires the parameter to read. An example is the parameter \textit{newHeight} in Figure 3.5. The instance of a locally declared variable can be read with the \textit{ReadVariableAction}, which requires the variable to read. An example of this is given in Figure 3.4, where the local variable \textit{surfaceArea} is read.

A literal value represents a constant value. In UML activities the node \textit{ValueSpecificationAction} is used to obtain the literal value. The node contains a constant value and puts this value on an output pin.

Returning a value is modeled in a similar way as reading the value of a parameter. In UML, the methods that return values have an additional parameter, which can only be used to return values. So instead of having an outgoing object flow, these \textit{ActivityParameterNodes} have an incoming object flow. An example is given in Figure 3.4, where the local variable \textit{surfaceArea} is returned.

Object creation creates an instance of a class. In UML activities this expression is mapped to the node \textit{CreateObjectAction}. This node requires the class it instantiates. The instance can be read from the output pin of the action.

The self reference is used to get the instance of the class which contains the method. In UML activities, this instance can be retrieved with the \textit{ReadSelfAction}, which puts the instance on an output pin.

The assignment to a global variable is mapped to UML activities with the \textit{AddStructuralFeatureValueAction}. This node requires the global variable to assign the value to, the instance of the class which contains the global variable, and new value of the global variable. The class instance is given with the \textit{ReadSelfAction} and the new value is put on another input pin. An example is given in Figure 3.5, where a new value is assigned to the global variable \textit{height}.

The assignment to a local variable is mapped to UML activities with the \textit{AddVariableValueAction} node. This node requires the variable to assign a value to and what value to assign. The latter is received by an input pin.

The method call calls the method of a specific class instance with a list of parameters. In UML activities the node \textit{CallOperationAction} is used to represent the method call. The node requires the method to call, the class instance the method is part of, and the list of parameters. The class instance is received by an input pin called \textit{target}. The parameters can be received by a list of input pins on the node. The node needs as many input pins as the method has arguments. In Figure 3.4 an example is given of the method \textit{*} (multiplication), which is called on an integer with another integer as parameter.

A more detailed mapping is given in Appendix C.

3.3.3 Interpretation of the flow graph

The mapping of data expressions to UML patterns is not the only part to consider to understand behavior modeled in UML activities. Section 2.2.2 introduced the interpretation of UML activities, which is extended in this section. This section describes definitions and properties to interpret the flow graph as a representation of imperative expressions.

In imperative programming languages like POOSL, expressions are executed in an order defined by control expressions. When an expression is executed, the next expression in the execution order is being executed.
The execution order in UML activities is determined with control tokens and data values traversing activity edges. The control expressions of POOSL are mapped to a structure of control flows in UML activities.

Figure 3.7 gives a schematic example of a UML activity. The node A is data dependent of B, because node A receives an object flow from B. The control flow indicates that node B has to be executed before node C.

The UML activity is interpreted as an imperative language. To get the expressions that are put in an execution order by control expressions, we remove the mapping of the control expressions in the activity. When this is done in Figure 3.7, the control flow is removed. Two connected subgraphs are left, namely node A and node B connected by the object flow, and node C by itself. These two subgraphs are the mappings of expressions that are not control expressions.

These connected subgraphs are nodes connected by object flows. Object flows are unidirectional. Cycles of activity edges are not allowed. The subgraphs are shaped like a tree, in which all the nodes have a path along edges to the root.

**Definition 1** A most dependent node is an action node or an object node in a UML activity from which has no object flows.

Definition 1 describes the root node of the connected subgraph. The subgraph has the following definition.

**Definition 2** A data connected component is a maximal subgraph in a UML activity where all the nodes are connected to the same most dependent node through a path of object flows, including this most dependent node itself. Every data connected component has exactly one most dependent node.

To show that these definitions can find the mapped expressions that are not control expressions in a UML activity, we regard the UML activity of Figure 3.4.

First we collect the most dependent nodes of the activity. These are the nodes $\text{AddVariableValueAction}$ on $\text{surfaceArea}$ and the $\text{ActivityParameterNode}$ on $\text{result}$. For each of the most dependent nodes, we collect the data connected component.

Figure 3.8 shows the most dependent nodes with a bold border. The nodes and object flows that are part of the data connected component are enclosed in a dashed border.

When a node received all control tokens and data values, it start executing. After this execution, all the nodes that are enclosed in the same border as the executed node are executed first. After the execution of all the nodes in the border, the control tokens are passed on to a node in a next border.
3.4 Conclusions

After mapping every single POOSL data class aspect to a UML pattern, we can draw several conclusions:

- We now know that UML can represent every aspect of POOSL data classes;
- The structural mapping from POOSL data classes to UML is straightforward;
- The mapping of behavior from POOSL data classes to UML activities is possible.

This chapter also described the interpretation of the UML activity flow graph. With this interpretation, the execution order of a UML activity represents the execution order of an imperative language.
CHAPTER 3. MAPPING OF POOSL DATA CLASSES TO UML
Chapter 4

Differences between the UML and POOSL metamodels

In this project we want to create a model-to-model transformation. The previous chapter discussed the POOSL syntax. This chapter introduces the POOSL metamodel, which is needed for a model-to-model transformation from UML to POOSL.

The transformation consists of transformation rules. In this chapter we identify the differences between the metamodels so that transformation rules can be defined in the next chapter. Only the differences of the behavioral part are identified, because the mapping of the structural part is straightforward.

4.1 The POOSL metamodel

The POOSL metamodel is based on the POOSL syntax, as described in Appendix A of the SHESim manual. The POOSL metamodel is created by Mathias Funk.

The POOSL metamodel differs slightly from the POOSL syntax. In the syntax of POOSL, sequential composition is an expression, namely $E_1; E_2$. In the POOSL metamodel sequential composition is modeled as a list of expressions.

4.2 Identifying differences

The differences between UML activities and POOSL data expressions are identified on the metamodel level. In this section we compare the UML metamodel and POOSL metamodel and identify the differences.

The structure of the UML and POOSL metamodels differ in two ways.

The first difference is the difference in hierarchy of metaclasses. The nodes and edges in UML activities are directly owned by an activity. The expressions in POOSL form a hierarchy. This hierarchy adheres to the recursive structure of the POOSL metamodel.

The second difference is that a UML activity is a graph and POOSL expressions are ordered in a list. Both structures represent the execution order of the model. The ordered list of expressions is a complete ordering.
CHAPTER 4. DIFFERENCES BETWEEN THE UML AND POOSL METAMODELS

Figure 4.1 shows a subset of the UML metamodel that describes the relation between activities, activity nodes and activity edges. The content aggregations, black diamonds, show that activity nodes and edges are owned by the activity. Activity nodes and edges are associated with one another through the properties \textit{source} and \textit{target}, where \textit{source} refers to the node the edge flows out of and \textit{target} refers to the node that the edge flows into.

Figure 4.2 shows a subset of the POOSL metamodel. A \textit{DataMethodDefinition} has a \textit{body} that is of the type \textit{Expressions}. The \textit{Expressions} metaclass owns \textit{Expression} metaclasses in an ordered list.

When sequential composition is modeled with the POOSL metamodel, the metaclass \textit{Expressions} is used. The ordered list of \textit{Expression} metaclasses represents the execution order of expressions.

The metaclass \textit{Expression} can contain other \textit{Expression} metaclasses, because of the recursive nature of the POOSL syntax. When a nonterminal in a meta expression is replaced by another expression, it is owned by the meta expression. The following example illustrates this structure.

Figure 4.3 shows a POOSL repository model. All the elements, like assignment and value, are instances of the metaclass \textit{Expression}. The repository model represents the expression \textit{a := 0} in POOSL. The assignment consists of the elements of the left-hand and the right-hand side. This is an example of the recursive nature of the POOSL metamodel.

Control expressions (except for sequential composition) are also modeled with a specialization of the \textit{Expression} metaclass. Control expressions can contain a sequence of expressions, like a branch of a conditional expression. This means that these specific types of \textit{Expression} metaclasses can hold the metaclass \textit{Expressions}, as shown in Figure 4.2.

4.3 Difference in hierarchy

The previous section showed that there is a difference in hierarchy of metaclasses between the UML and POOSL metamodels. This section elaborates on the difference of structure for data dependencies and each control expression.

For data dependencies and each control expression an example is given of a UML activity containing the mapping of such an expression. The structure of the UML activity in the UML
4.3. DIFFERENCE IN HIERARCHY

The difference in hierarchy is that in the UML repository model node \(A\) and node \(B\) are directly owned by the activity, and in the POOSL repository model \(B\) is owned by \(A\), because \(A\) is data dependent on \(B\).
4.3.2 Sequential composition

Figure 4.6 shows a UML activity and repository model. There is a sequential composition between two data connected components. One data connected component consists of node A and node B, where node A is data dependent on node B. The other data connected component consists of node C.

The behavior modeled in Figure 4.6 is shown in Figure 4.7 as a POOSL repository model.

The difference is that in the UML repository model all nodes are at the same level, and in the POOSL repository model the expressions in sequential composition at the same level, their data dependencies are owned by these expressions.

4.3.3 Conditional expression

Figure 4.8 show a UML activity and its UML repository model. The activity shows the mapping of a conditional expression. When the condition c is satisfied, the branch with the nodes A and B is followed. Otherwise the branch with the node C is followed.

Figure 4.9 shows the conditional expression of Figure 4.8 as a POOSL repository model.

The difference between both repository models is caused by the fact that in UML all nodes are on the same level whereas in POOSL the nodes are ordered according the the underlying metamodel.
4.3. DIFFERENCE IN HIERARCHY

4.3.4 Loop expression

Figure 4.8 shows a UML activity and its UML repository model. The activity shows the mapping of a loop expression. As long as condition $c$ is satisfied, node $A$ is executed and then node $B$. The repository model shows that node $A$ and node $B$ are part of the body. The ValueSpecification-Action is part of the test property of the loop node, meaning that it models the condition. The property decider of the loop node reads the validation of the condition $c$ from the output pin of the ValueSpecificationAction. Note that the nodes that are in the property test of loop node are not visible in the UML activity.

The POOSL repository of Figure 4.11 has a similar structure as the loop node in UML. The only difference is that the body of the LoopExpression is modeled as sequential composition with the Expressions metaclass, and the body of the loop node has all nodes and edges at the same level. The difference in sequential composition in the body of the loop expression is identical to the difference in sequential composition mentioned earlier.

Figure 4.8: A UML activity with a conditional structure and its repository model

Figure 4.9: The POOSL repository model of the translated UML activity in Figure 4.8
4.4 Conclusions

In this chapter we identified the differences between UML activities and POOSL expressions on metamodel level. The metamodels differ in the following two ways:

- The hierarchies of the metaclasses are different.
- The execution order is defined as a graph in the UML metamodel and as a list in the POOSL metamodel.

For the difference in hierarchy, we observed data dependencies and the control expressions in detail.
Chapter 5

Transformation from UML to POOSL

The previous chapters introduced the languages UML and POOSL. Through a mapping, as described in Chapter 3, we regarded the differences of the structural part and the behavior of the languages.

Chapter 4 described the differences between the UML metamodel an the POOSL metamodel. In this chapter we create transformation rules to reconcile the differences and translate a UML model in a POOSL model. An example is given afterwards.

The transformation has been implemented and is described in Section 5.5.

5.1 Overview

Figure 5.1 shows a schematic overview of our transformation. The UML model consists of UML classes and UML activities, which represent the modeled system. By using a model-to-model transformation, the UML model is transformed into a POOSL model, which is an instance of the POOSL metamodel. After this step, the POOSL model is transformed into a textual version, namely the POOSL code. The POOSL metamodel and the model-to-text transformation from a POOSL model to POOSL code are made by Mathias Funk.

This chapter only describes the model-to-model transformation. The transformation rules are defined on the metamodel level between the UML metamodel and the POOSL metamodel.

First we look at the transformation itself. The transformation of the structural part is described in Section 5.2. The transformation of the behavior is described in Section 5.3. An example of the transformation is shown in Section 5.4. The tools that are used and considered for use are introduced in Section 5.5.

Figure 5.1: Overview of the transformation
CHAPTER 5. TRANSFORMATION FROM UML TO POOSL

5.2 Transforming the structural part

The structural part in POOSL is described with the data class definition and the data method definition. In UML the structural part is described with UML classes package. For the transformation we want to know which metaclasses in the UML metamodel map to which metaclasses in the POOSL model. This is very similar to the mapping of the structural part in Section 3.2.

Figure 5.2 shows the class definition and method definition in the UML metamodel and the POOSL metamodel. The transformation of the structural part from a UML model to a POOSL model is done by locating the aspects in both metamodels.

To show the transformation of the structural part from UML to POOSL, the POOSL aspects of Section 3.2 are chosen. For every aspect the reference in the UML metamodel and the reference in the POOSL metamodel is shown. The reference refers to a metaclass in the metamodel and a property of that metaclass, to show where the information of the aspect is located.

Table 5.1 shows the locations of the aspects of the class definition and Table 5.2 shows the locations of the aspects of the method definition.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>UML location</th>
<th>POOSL location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class name</td>
<td>Class.name</td>
<td>DataClassDefinition.name</td>
</tr>
<tr>
<td>Class inheritance</td>
<td>Class.superClass</td>
<td>DataClassDefinition.superclass</td>
</tr>
<tr>
<td>Instance variables</td>
<td>Class.ownedAttribute</td>
<td>DataClassDefinition.instanceVariables</td>
</tr>
<tr>
<td>Instance methods</td>
<td>Class.ownedOperation</td>
<td>DataClassDefinition.methods</td>
</tr>
</tbody>
</table>

Table 5.1: Table of data class definition

<table>
<thead>
<tr>
<th>Aspect</th>
<th>UML location</th>
<th>POOSL location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name</td>
<td>Operation.name</td>
<td>DataMethodDefinition.name</td>
</tr>
<tr>
<td>Parameters</td>
<td>Operation.ownedParameter</td>
<td>DataMethodDefinition.parameters</td>
</tr>
<tr>
<td>Return type</td>
<td>Operation.type</td>
<td>DataMethodDefinition.returnType</td>
</tr>
<tr>
<td>Local variables</td>
<td>Activity.variable</td>
<td>DataMethodDefinition.localVariables</td>
</tr>
</tbody>
</table>

Table 5.2: Table of data method definition
The transformation of the structural part can be implemented with this straightforward mapping of metaclasses from the UML metamodel to metaclasses in the POOSL metamodel.

5.3 Transforming behavior

This section describes the transformation of behavior. Chapter 4 identified the differences between the UML metamodel and the POOSL metamodel. In this section we reconcile the differences with transformation steps.

Instead of transforming the UML activity immediately into POOSL data expressions, the transformation consists of three steps:

- Grouping patterns, which alters the hierarchy of the UML activity to the hierarchy of the POOSL metamodel;
- Ordering groups, which transforms the UML activity graph to a chain with complete ordering, just like the ordered list of expressions in the POOSL metamodel;
- The transformation from UML activities to POOSL data expressions.

The following sections discuss each of these steps.

5.3.1 Grouping patterns

Chapter 4 elaborated on the difference in hierarchy of the UML metamodel and the POOSL metamodel. The data dependencies and the control expressions were analyzed.

In this section we change the hierarchy of a UML activity in such a way that it becomes similar to the hierarchy of the POOSL metamodel. This is done by grouping nodes and edges together, hence the name grouping. These nodes and edges that are grouped together are put in a structured activity node, which is a node in UML activities that can contain other nodes and edges. The notation for structured activity nodes is a dashed border around the edges and nodes it contains.

The structure of data dependencies is similar in both metamodels, by comparing the POOSL metamodel with the UML activity instead of the UML metamodel. For this reason we only reconcile the differences of the control expressions between the metamodels.

Sequential composition

In Section 4.3.2 an example of sequential composition was shown in both a UML and a POOSL repository model. The difference between the two models is that in UML activities all nodes and edges are at the same level, and in the POOSL repository model the expressions of sequential composition are at the same level.

To reconcile this difference, the separation of expressions in sequential composition should become visible in the UML repository model. This separation is achieved by putting all data connected components in separate structured activity nodes.

Figure 5.3 shows the UML activity of Figure 4.6 after grouping. Every structured activity node represents an expression in sequential composition. Note that the control flow now flows between these nodes.

Conditional expression

In Section 4.3.3 an example of a conditional expression was shown in both a UML and a POOSL repository model. The difference between the two models is that all the nodes and edges of the UML activity are at the same level, and in POOSL expressions the expressions adhere to the grammatical structure.
CHAPTER 5. TRANSFORMATION FROM UML TO POOSL

The difference can be reconciled by grouping the whole conditional expression in a structured activity node. To give the conditional expression in UML activities the same structure as in the POOSL metamodel, the branches need their own structured activity node. This represents the \textit{Expressions} metaclass in the POOSL model, which model the branches. These branches use the grouping of the sequential composition mentioned above.

Figure 5.4 shows the UML activity of Figure 4.8 after grouping. There is no merge node, because after executing one branch there are no control tokens to pass on within the structured activity node.

\textbf{Loop expression}

In Section 4.3.3 an example of a loop expression was shown in both a UML and a POOSL repository model. There is no structural difference, but the body that is in sequential composition has to be treated as such.

All structures in the UML activity that do not represent sequential composition are part of a structured activity node. For the sake of consistency in the UML activity, the loop node is also put in a structured activity node.

The differences in the UML activity are shown in Figure 5.5.
5.3. TRANSFORMING BEHAVIOR

Figure 5.5: A UML activity with a loop node and its repository model after grouping

Figure 5.6: Restriction of the UML metamodel

Result of grouping

Grouping collects all nodes and edges that represent expressions in sequential composition and put them together in a structured activity node. The structured activity nodes are connected by control flows to imply the execution order. The model still conforms to the UML metamodel. Nevertheless the new model is constrained. Instead of that the activity owns all kinds of nodes and edges directly, it only contains structured activity nodes and control flows. The structured activity nodes now contain all the nodes and edges of the activity before grouping. This restriction is shown in Figure 5.6.

5.3.2 Order groups

After grouping the UML activity, the flow graph only consists of structured activity nodes and control flows. The execution order of the activity is given by control flows, flowing from one structured activity node to another. This composition is called a network. An example of a network is given in Figure 5.7. In the POOSL metamodel the expressions are in an ordered list. The structured activity nodes connected by control flow are not necessarily in a complete order like the ordered list. This transformation step makes sure that the UML activity obtains a complete order of execution. This is done by transforming the network of nodes to a chain of nodes. An example of a chain of nodes is shown in Figure 5.8.

The first part of the transformation describes an algorithm that transforms the activity graph into an ordered list of nodes. The list is ordered according to the execution order of the nodes. It is possible that several orders comply to the execution order. The algorithm takes one of these options.
The algorithm transforms a graph of nodes ($N$), that initializes with the nodes of the activity, into a list of nodes ($N'$), that starts off being empty. Nodes have the property of having preceding and following nodes. Before constructing the algorithm, several cases should be observed.

1. A node has only following nodes.
2. A node has only preceding nodes.
3. A node has preceding and following nodes.

In the first case (like node 1 and 5 in Figure 5.7), the present node may be executed, because it has no predecessors. This means that the present node may be added to $N'$ and the algorithm should continue with all following nodes.

In the second case (like node 4 in Figure 5.7), the preceding nodes are checked if they are already added to the list of nodes $N'$. If there is at least one preceding node where this property does not hold, the algorithm should first start processing this node.

In the third case (like node 2 and 3 in Figure 5.7), the procedure at case 2 should be followed first and then, if the present node is not in $N'$, the procedure at case 1.

We have a recursive function $f$ of type $\mathcal{R} \rightarrow \mathcal{L}_\infty(\mathcal{N})$, where $\mathcal{L}$ is the list type and $\mathcal{N}$ the type of nodes. The function initializes with a node without preceding nodes. The cases mentioned above are translated formally into rules $A1$...$A4$. Every time the function $f$ is called upon a node, it starts off checking rule $A1$. If the condition of $A1$ holds, the rule is used, or else the function proceeds with $A2$, and so forth.

A1. $f.n = f.p$, if there is a preceding node $p$ for which $p \not\in N'$;
A2. $f.n = []$, if $n \in N'$;
A3. $f.n = n >> f.q_1...f.q_m$, where $q_1...q_m$ are all following nodes of $n$;
A4. $f.n = [n]$, if $n$ has no following nodes.

In Figure 5.7 an example is given of a graph of nodes. To show function $f$ in action, the function is called on node 5 in the example graph.
Figure 5.8: The chain of the network, shown in Figure 5.7

\[ f.5 = \{\text{Rule A3}\} \]
\[ 5 \triangleright f.4 \]
\[ f.4 = \{\text{Rule A1; choice between 2 and 3}\} \]
\[ 5 \triangleright f.2 \]
\[ f.2 = \{\text{Rule A1}\} \]
\[ 5 \triangleright f.1 \]
\[ f.1 = \{\text{Rule A3}\} \]
\[ 5 \triangleright 1 \triangleright f.2.f.3 \]
\[ f.3 = \{\text{Rule A3}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright f.4.f.3 \]
\[ f.4 = \{\text{Rule A1}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright f.3.f.3 \]
\[ f.3 = \{\text{Rule A3}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright 3 \triangleright f.4.f.3 \]
\[ f.4 = \{\text{Rule A2}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright 3 \triangleright 4 \triangleright [] \]
\[ f.5 = \{\text{Property of \triangleright}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright 3 \triangleright [4] \]
\[ f.4 = \{\text{Property of \triangleright}\} \]
\[ 5 \triangleright 1 \triangleright 2 \triangleright [3, 4] \]
\[ f.3 = \{\text{Property of \triangleright}\} \]
\[ 5 \triangleright 1 \triangleright [2, 3, 4] \]
\[ f.2 = \{\text{Property of \triangleright}\} \]
\[ 5 \triangleright [1, 2, 3, 4] \]
\[ f.1 = \{\text{Property of \triangleright}\} \]
\[ 5 \triangleright [5, 1, 2, 3, 4] \]

The result is a list of nodes that show one of the possible orders the nodes can be executed.

The second part of this transformation step transforms the list of nodes ordered according to the execution order into a chain of nodes.

This part of the transformation takes the first node of the list and creates a control flow to the following node in the list. This is repeated until the last node, which has no outgoing control flows. When this part is performed on the example that was used in the first part, the chain shown in Figure 5.8 is the result of this part.

This transformation step is performed on the activity, the branches of the conditional expression and the body of the loop node.
5.3.3 From UML activities to POOSL expressions

The last two transformation steps transformed the activity of the UML model in such a way that the hierarchy and ordering is comparable to expressions in the POOSL metamodel. This section describes the last transformation step which transforms the grouped and ordered UML activity into a POOSL model.

This last step depends mostly on the mapping described in Chapter 3. This section does not describe the POOSL dependent transformation, but the approach to perform this transformation.

In this transformation the list of expressions owned by the metaclass Expressions, that represents the body of a method in the POOSL metamodel, is constructed. This is done in the order of execution. After the previous transformation step, the UML activity is a chain ordered by the execution order. The algorithm of this transformation step takes the first node of the activity and translates its contents to POOSL. This is repeated for every node in the chain.

The nodes in the chain of the UML activity are structured activity nodes that contain the nodes and edges that represent a POOSL expression. There are three kinds of compositions of nodes and edges in the structured activity nodes that can be expected:

- A data connected component;
- The mapping of a conditional expression where the branches are represented by structured activity nodes;
- A single loop node, representing a loop expression.

Each of these compositions need a different transformation method.

The algorithm of this transformation step is performed as follows:

algorithm: transformActivityToExpressions
input: \((N, E)\), the first node of the grouped and ordered activity
output: expressions, an instance of the Expressions metaclass

01: for each \(n \in N\) in execution order

02: if \(n\) contains nodes and object flows

03: \(mdn :=\) Call the most dependent node of \(n\)

04: expressions.add(transformToPOOSL(mdn))

05: if \(n\) contains a if structure

06: Create ConditionalExpression as \(ce\)

07: \(thenFlow :=\) the control flow with the guard

08: \(thenNode :=\) the node that represents the then branch

09: \(elseNode :=\) the node that represents the else branch

10: \(ce.addCondition(transformToPOOSL(thenFlow.guard))\)

11: \(ce.thenClause := transformActivityToExpressions(thenNode)\)

12: \(ce.elseClause := transformActivityToExpressions(elseNode)\)

13: expressions.add(ce)

14: if \(n\) contains a loop node

15: \(ln :=\) the loop node \(ln\)

16: Create LoopExpression as \(le\)

17: \(le.addCondition(transformToPOOSL(ln.decider))\)

18: \(le.body := transformActivityToExpressions(ln.bodyPart)\)

The algorithm has the nodes and edges as input. The output is the instance of the Expressions metaclass that represents the body of the POOSL data method.

The algorithm scans the structured activity nodes of the activity in execution order. For each of the nodes, one of the three composition of nodes and edges can be expected.

A structured activity node can contain a data connected component. In this case the most dependent node is transformed to the equivalent in POOSL with the function transformToPOOSL.
5.4. EXAMPLE

This function takes a node and transforms it to the POOSL equivalent. Afterwards, the function recursively translates all the nodes the input node is data dependent of into a POOSL equivalent.

If the structured activity node contains an if structure, the specialized POOSL expression `ConditionalExpression` is created. This `Expression` metaclass requires a condition, a then clause and an else branch. The condition is the translated guard of the control flow that holds the guard. The then and else clause are modeled each as an `Expressions` metaclass. This algorithm is used on the structured activity nodes that represent the branches, since these are also modeled as a collection of nodes and edges that represent sequential composition.

When the structured activity node contains a loop node, the specialized expression `LoopExpression` is modeled. This `Expression` metaclass requires a condition and a body. The condition is the translated condition of the loop node. The body in POOSL is modeled with the `Expressions` metaclass. This algorithm is used on the body property of the loop node, since it is also modeled as a collection of nodes and edges that represents sequential composition.

When all nodes of the UML activity have been translated, the output is an instance of the `Expressions` metaclass that has a list of expressions, which are the translated nodes and edges modeled in the structured activity nodes in the UML activity.

5.4 Example

To illustrate how the transformation in Section 5.2 and Section 5.3 works, this section shows an example. The example is a class taken from the default POOSL data classes. The context of this class is explained first. After that the class is modeled in UML and finally transformed.

5.4.1 The class Dictionary

One of the default data classes in POOSL is the class `Dictionary`. This data class describes a data structure that models a list of elements. Every element has a key and an object it stores. When we have a key and we want to retrieve the object that is identified by it, we use the method `at` which takes the key as input and the has the object as returning value. This is the method that we will model in this example.

We only model a part of the class `Dictionary`. We only model the structural part of the class and the method `at`. This model is transformed into a POOSL model.

5.4.2 Modeling the Dictionary class in UML

First we model the structural part of the class `Dictionary` to UML classes. This part of the UML model is shown in Figure 5.9.

The list of elements is modeled as two arrays, where the array `KeyList` contains all the keys of the elements, and `ValueList` contains all the objects of the elements.

Next to the method `at` that we are going to model in UML activities, there is a method called `find`. This method returns the index of the element with the input key. This method is used in the method `at`.

The class `Dictionary` inherits the properties of the class `Object`. The class `Dictionary` is not fully modeled. Only the elements that are necessary to model the behavior of the method `at` are modeled.

Secondly, we model the behavior of the method `at` of the class `Dictionary` to UML activities. This part of the UML model is shown in Figure 5.10.
When we want to retrieve an element out of the array with a specific key, we want to know if this key actually exists. In the UML activity we have a local variable called `Index` that will contain the index of the key in the dictionary. This index is retrieved by using the method `find`. If the result is 0, the key does not exist in the dictionary. This result is written to the local variable `Index`. In an `if` structure, we see if the method the index is 0 or not. When the index is 0, we return the constant value `null`. If the index is not 0, the key exists in the dictionary. The object that is at the index of the key is returned. We do this by calling the method `get` on the array `ValueList` that contains all the objects.

5.4.3 Transforming the UML model to a POOSL model

In this section we look at the transformation of the UML model to the POOSL model. First we describe the transformation of the structural part of a class, and afterwards to the transformation of behavior.
5.5. IMPLEMENTATION

Structural part

The structural part is transformed with a mapping described in Section 5.2. The UML class of Figure 5.9 is mapped to the POOSL model as shown in Figure 5.11.

Behavior

The behavior in the UML activity is transformed with the three transformation steps defined in Section 5.3.

The first transformation step is grouping. First we group data dependencies in sequential composition. It is done by collecting all the most dependent nodes and collect the data connected components and put them in structured activity nodes. The result of this grouping is shown in Figure 5.12, where all the bold bordered nodes are the most dependent nodes. Figure 5.13 shows the activity after grouping the conditional expression as well.

The second transformation step orders the structured activity node. In this example there are only two structured activity nodes in a chain. The transformation is not necessary. To see an extended example of the second transformation step, see Section 5.3.

The third and last transformation step is the transformation of the UML activity to a POOSL model. The POOSL model of the behavior is shown in Figure 5.14.

In the UML activity are two structured activity nodes. In the first structured activity node in the activity is a data connected component. According to the algorithm we take the most dependent node. This node indicates an assignment, as shown in the POOSL metamodel. The left-hand side is the variable in the node and the right-hand side is derived by following the object flow in the opposite direction and translate every single node to POOSL.

The second structured activity node contains an if structure so a ConditionalExpression is created in the POOSL model. The condition of the ConditionalExpression is derived from the control flow with the guard. The structured activity nodes that represent the then and else branches are translated into POOSL.

5.5 Implementation

The previous sections described the transformation steps to transform a UML model to a POOSL model and saw an example. This section describes the aspects of implementation. First we discuss the tools that were used and considered and afterwards we discuss how the development
Figure 5.12: The activity of the method at after grouping data connected components

Figure 5.13: The behavior of the method at after grouping
5.5. IMPLEMENTATION

Figure 5.14: The behavior of the method `at` in the POOSL metamodel

was experienced. The validation of the implementation is given in Appendix D. This appendix makes use of the example of Section 5.4.

5.5.1 Tools

To implement a model-to-model transformation, a tool must be used that supports it. In the search for tools that have this property and work within the Eclipse framework, the following two tools have been found:

- xTend of openArchitectureWare[9];

In this project, xTend of openArchitectureWare is used. The motivation for this choice is the available support. The site of openArchitectureWare[22] has an active forum and thorough documentation. Besides that, the works of Funk, as mentioned in Section 5.1, were implemented with the tools of openArchitectureWare.

An example of an xTend transformation in practice is shown in a document of Voelter[23]. It shows how an environment should be configured, how to define a custom metamodel and an example of a model-to-model transformation from a UML model to the custom model.

5.5.2 Development

The implementation time of this transformation was approximately a week. This was after getting familiar of the tools in the previous section. The difficulties of the development were in understanding the POOSL metamodel. As shown in the example of this chapter, there are several elements that have to be modeled to conform to the POOSL metamodel, like the element `PrimaryExpression`. This metamodel was not suited for model transformation.
A model-to-model transformation is rapidly developed with the transformation method described in this chapter. After finishing the transformation from UML models to POOSL, we decided to transform to other languages as well. So we tried to transform the UML model to a JAVA model. Although we only partially implemented the transformation to JAVA, within several days we were capable of transforming the UML models to JAVA models that could represent the same expressions as we have seen in POOSL expressions.

In conclusion we can say that our method of model-to-model transformation is easy to implement, when the developer is familiar with the metamodel of the new transformation method.
Chapter 6

Conclusions

We implemented the transformation from UML to POOSL in an Eclipse environment during this project. The transformation has been tested on UML models that represent predefined POOSL data classes and our small test cases. This proves that it is possible to model behavior with UML activities that can be used for code generation.

In this project we created an interpretation for UML activities. The interpretation is not POOSL related, but interprets the activity as an imperative program.

Not all transformation steps are POOSL related. There are two transformation steps to structure the UML activity for interpretation as an imperative program.

During this project we built a transformation from UML to Java as well. For this transformation we used all transformation steps except for the POOSL specific step. This step was replaced by a Java specific transformation. This shows that the interpretation of a UML model is not POOSL specific.
Chapter 7

Further work

During this project, several question were raised. They provide interesting topics for further research.

First of all, we considered there should be a better way of representing UML activities. We looked for possibilities of using a surface language, which is a language that represents a part of the activity. Defining a surface language that support the semantics of activities would prove to be useful to reduce the number of nodes and edges in a UML activity.

An interesting observation is that in the search for articles on transformations from UML to any other language, behavior in UML is often modeled with state machines. As a matter of fact, articles on the semantics of UML actions in UML activities, are rare in general. An interesting field of research would be what advantages and disadvantages each behavior modeling technique has by comparing them.

When modeling instances of UML activities, there were some aspects that can only be modeled in a static way. When we look at the guards on edges, for defining a conditional expression, they cannot be modeled with actions, but only with constant values. This would mean that for model transformations you always need to build a parser to translate the literal value into another language. A solution that would be able to model these guards (and other static values) with elements offered by UML makes a parser for the transformation obsolete.

Another interesting topic is about basic types and operators. UML 2.0 defines several primitive types, like strings and integers, but does not declare operators for them. In this project the lack of operators was solved by declaring an operator as an operation of a class that defines the type, as shown with multiplication and integers in the example of Section 3.3.1.
Chapter 8

Related Work

Model driven engineering is a technology that has a broad field of research. One of the challenges in this field is finding a modeling technique that can be used for different kinds of design.

In this project we want UML models not to have POOSL specific elements. We restricted the interpretation of UML models to a model of imperative object-oriented languages. Florescu et al.[14] use a profile for UML, so that POOSL can be integrated in the model. UML models built with this profile contain POOSL specific elements.

Our interpretation of UML activities is one of the many that can be chosen. The UML specification[21] does not specify the semantics of UML activities fully. Bock[4] even says literally about token movement rules in a UML activity:

"It is hoped that the rules are precise enough to be translated to a formal semantics, especially to support proving properties about modeled processes. This is left for future work."

Research for defining formal semantics for UML activities has been committed by several parties, like the formalization of Eshuis et al.[12].

We chose model-to-model transformation in the project. This is based on one of the interpretations of model driven engineering, namely Model Driven Architecture (MDA)[19]. In MDA a platform independent model is transformed to a platform specific model.

Model transformations are used in many projects. A taxonomy of model transformations has been proposed by Van Gorp et al.[17]. With this taxonomy we can classify model transformations. The model transformation of this project is a horizontal exogenous model transformation. This means that the transformation from the source and target model is at the same level of abstraction, and that the transformation is performed between models expressed in different languages.

We give an example of two projects that also use model transformations, and compare it to our project.

One of these projects is a project within the Ideals project that was performed to migrate models from a legacy architecture into a new architecture[8]. The migration is a model-to-model transformation from a finite-state machine to a paradigm based on task-resource systems. The transformation has the same classification as our transformation, but it is performed in a migration context. Our project at the other hand uses transformations for validating the model in the target language.

Another project that also uses model transformations within the same classification is the SmartHome project of Siemens. In this project Groher et al.[18] researched how variations in abstraction can be implemented in model-to-model transformations. These model-to-model transformations transform an abstract model to a concrete model. This means that the transformation is vertical in stead of horizontal.
CHAPTER 8. RELATED WORK
Bibliography


Appendix A

Data Class Definition

The syntax of a POOSL data class consists of three levels. This document maps the first level to UML patterns. The first level describes the interface of the POOSL data class. The interface contains the data method declarations. The syntax of the data method declarations is the second level in the POOSL syntax.

POOSL data class definitions are defined in the following Backus-Naur form:

\[
CD = \text{data class } C \[ \text{extends } C_{\text{super}} \]
\text{instance variables } x_1 \ldots x_n
\text{instance methods } MD_1 \ldots MD_k,
\]

where \( CD \) is the data class definition, \( C \) the class name, \( C_{\text{super}} \) the class to inherit from, \( x \) an instance variable and \( MD \) a data method definition.

POOSL data classes contain three aspects that should be modeled in UML to be able to perform the mapping:

- Inheritance
- Instance Variables
- Instance Methods

POOSL data classes are mapped to the UML metaclass \( \text{Class} \). From there the other aspects are mapped intuitively, making the mapping in this chapter straightforward. Every single aspect individually is mapped to a UML pattern in the following sections. The last section of this chapter will give an example of a mapping, where all the mentioned aspects will be present.

A.1 Inheritance

A POOSL data class can inherit from one other data class with the keyword \text{extends}. This means that the present class inherits the instance variables and instance methods of the parent.

The way to model inheritance in a UML class diagram is via the property \( \text{superClass} \). This property is derived from generalization associations. These associations refer to another class, which is the parent. This is sufficient to model POOSL data class inheritance in UML.

Figure A.1 shows the involved elements in the UML metamodel. It shows the metaclass \( \text{Generalization} \) and the derived \( \text{superClass} \) association. The metaclass \( \text{Generalization} \) has two associations with \( \text{Class} \). The association labeled \text{general} refers to the \( \text{Class} \) that is the superclass and the association labeled \text{specific} refers to the \( \text{Class} \) that inherits from the superclass.
APPENDIX A. DATA CLASS DEFINITION

The cardinality of the `superClass` association is different from the UML standard. In UML there are several superclasses for every class. In POOSL however this number is 1 or 0.

A.2 Instance Variables

A POOSL data class can contain variables that can be used by all methods owned by that class. These are called instance variables.

Every UML Class can contain a list of attributes. These are represented by the metaclass `Property`. Like POOSL instance variables the name and type of the UML `Property` class can be set. Figure A.2 shows the required elements in the UML metamodel.

A.3 Instance Methods

A POOSL data class can contain methods which can be called once the class is instantiated. These are called instance methods. A method in a POOSL data class has a name, zero or more parameters and a result.

A UML class can own a list of methods. In UML, a method is represented by an instance of the `Operation` metaclass. An `Operation` class has a name, a return `Type` and a list of `Parameter` classes.

Figure A.3 shows what UML elements are required in the UML metamodel. In POOSL it is mandatory for data methods to have a result value. In UML this resulting value is considered to be a `Parameter` (with `direction = return`).

In a mapping from UML to POOSL it should be noted that in UML result parameters are not mandatory. If this parameter is missing the instance of the POOSL data class should be returned when transforming the method into POOSL.

A.4 Data class example

In this example a class `TestClass` is created. It has an instance variable called `buffer` of type `Number`. The value of `buffer` can be read with the method `getBuffer` and can be written with the method `setBuffer`. The class inherits the `Object` class.
The code of TestClass is displayed below. The implementation of the methods is not relevant now and will not be modeled in this example.

When the TestClass is mapped to a UML class diagram, it looks like Figure A.4. Another interesting view is what metaclasses in the UML metamodel are being used. When a metaclass is being used, it is instantiated in the UML model. An instantiation of a metamodel is called a repository model in UML. The UML repository model is shown in Figure A.5.

```plaintext
data class TestClass
extends Object
instance variables
    buffer: Number
instance methods
    setBuffer (in: Number): TestClass
        buffer := in.
    getBuffer (): Number
        return buffer.
```
Figure A.5: The UML repository model of the TestClass example
Appendix B

Data Method Definition

In the data class definition is a reference to the POOSL data method definition. The syntax of the data method definition is the second level of the POOSL data class syntax, which will be mapped to UML patterns in this chapter. The data method definition describes the interface of a data method.

POOSL data method definitions are defined in the following Backus-Naur form:

\[
MD = m(u_1, \ldots, u_n) \ R \\
\hspace{0.5cm} | z_1 \ldots z_m | \\
\hspace{0.5cm} E \\
\hspace{0.5cm} | m(u_1, \ldots, u_n) \ R \\
\hspace{0.5cm} | z_1 \ldots z_m | \\
\hspace{0.5cm} \text{primitive.}
\]

where \(MD\) is the data method, \(m\) the method name, \(z\) a local variable, \(u\) a parameter, \(R\) a result type of the data method and \(E\) is a data expression (which is the last level of the POOSL data class syntax).

The data method definition is the specification of the actual behavior of a method. In the first level of the POOSL syntax, POOSL data methods are mapped to the UML metaclass \textit{Operation}. The third level of the POOSL syntax describes the behavior of a method and will be mapped on the UML metaclass \textit{Activity}. To connect the POOSL data method definition and its behavior in the mapping to UML, the association \textit{specification} between \textit{Activity} and \textit{Operation} will be used.

The formal syntax of the data method definition shows two options. The first option is that the method consists of POOSL data expressions. The second option is that the method is \textit{primitive}. This means that the method body is provided by the semantics of POOSL.

Both options can be mapped in the UML. The first option will have an \textit{Activity} in the UML model and the second option has no \textit{Activity} in the UML model.

Note that there are no operation headers in POOSL. When the method is introduced, the body is given immediately. POOSL also does not describe abstract methods.

This chapter describes the mapping from POOSL to UML of these three aspects of POOSL data methods:

- Parameters
- Result
- Local Variables

Since the \textit{Activity} will contain the behavior of the data method, it should be possible that it can access all aspects of the data method definition. This chapter will put focus on the accessibility of the aspects by the \textit{Activity}.
APPENDIX B. DATA METHOD DEFINITION

B.1 Parameters

When calling a method in POOSL, it is possible to pass on variables to add to the data method context. These variables are called parameters. The Operation metaclass has a property ownedParameter that refers to instances of the UML Parameter class. In order for the Activity of the method to make use of these parameters, the Activity must have the same ownedParameters declared as the Operation. In Figure B.1 the relation between Operation and Activity is shown in the UML metamodel. Activity has an association with Operation, which is called specification. Activity and Operation both have the property ownedParameter. They refer to two physically different lists of Parameters, but in order to conform to the UML metamodel, they should have the same Parameters in the same order.

B.2 Result

Every method in a POOSL data class has a result value. This is the value, that is being returned by the method. In POOSL it is not mandatory to specify the value to be returned. If this is not specified, the method will return a value of its own class type. The result value is also a parameter in UML. Hence it works in the same way as in Section B.1. The only exception is that instead of the default value of direction, which is in, in Parameter, it is set to return.

B.3 Local Variables

It is possible in POOSL to declare local variables in a data method. Local variables are variables that can only be used in the scope of the method where they are declared. Local variables are part of a method’s body, so they are associated with the UML Activity in order to be accessed directly by the Activity. In Figure B.2 the location of the local variables in the UML metamodel is shown. In this case the local variables can be represented as instances of the metaclass Variable classes. An alternative is that instead of using Variables in Activity, the local variables can be stored as attributes of the Activity in Property. Activity can have attributes, because it inherits from Class that supports attributes.
Figure B.2: Association Variable and Activity in UML Activities
Appendix C

Data Expressions

In the syntax of the data method definition is a reference to the POOSL data expressions. The syntax of the data expressions is the third and last level of the POOSL data class syntax. Data expressions describe the behavior of the data methods.

POOSL data expressions are defined in the following Backus-Naur form:

\[
E = \begin{align*}
& x & \quad \text{global variable} \\
& u & \quad \text{local variable} \\
& \gamma & \quad \text{literal} \\
& \text{return } E & \quad \text{return} \\
& \text{new}(C) & \quad \text{object creation} \\
& \text{self} & \quad \text{self} \\
& E \ m(E_1, ..., E_n) & \quad \text{dynamic method call} \\
& E \ m_C(E_1, ..., E_n) & \quad \text{static method call} \\
& x := E & \quad \text{assignment to global variable} \\
& u := E & \quad \text{assignment to local variable} \\
& E_1;E_2 & \quad \text{sequential composition} \\
& \text{if } E_c \text{ then } E_1 \text{ else } E_2 \text{ fi if} \\
& \text{while } E_c \text{ do } E \text{ od while}
\end{align*}
\]

Every data expression has its own section in this chapter.

In this document we will map behavior in POOSL on the UML metaclass Activity.

An UML Activity contains Actions with directed edges in between. There are two types of directed edges (represented by ActivityEdge), namely ControlFlow and ObjectFlow. The former specifies the order in which actions should be executed and the latter passes entities through the flow. The ObjectFlow does not connect Actions directly; it connects OutputPins and InputPins that are associated with an Action. An ObjectFlow sends instances from the source to the target.

This chapter will also describe a way Actions could be represented by visual notation in UML. An Action with specific semantics is represented by having the Action name stereotyped (enclosed in ≪ and ≫). When this Action needs a reference to an operation, Type or Variable, it is written in the Action itself without being stereotyped.

The chapter will also show how the number of Actions in a UML activity diagram can be reduced. This is done by using OCL expression, which will be explained in Section C.1.
APPENDIX C. DATA EXPRESSIONS

C.1 Using OCL expressions in UML

The *Object Constraint Language* (OCL) can be used to create references to metaclasses in the UML metamodel. In the scope of this document, OCL will be used to replace several *Actions* with an OCL expression, to reduce the number of *Actions* in a UML activity diagram and to implement guards.

The OCL specification describes what needs to be added in the UML metamodel to be able to implement OCL expressions. OCL requires several new metaclasses, which are shown in gray in Figure C.1. In plain UML, there is an metaclass called *OpaqueExpression*. The metaclass *ExpressionInOCL* inherits from this metaclass. The attribute *language* is inherited and must have the value set to "OCL". The OCL expression should be recorded in *OCLExpression* via the association *bodyExpression*.

OCL expressions require a context. The context is referred to with the association *contextVariable*. According to the OCL specification this context is the *contextual classifier*: the namespace in which the expression is evaluated. In the expression the instance of the *contextual classifier* can be referred to with the keyword *self*.

In this chapter, where OCL expressions will be connected to nodes, the *Activity* is always the *contextual classifier*. So all *self* keywords in the OCL expressions refer to the *Activity*. All the OCL expressions given are at the moment not verified.

C.2 Global Variable

POOSL global variables are the instance variables of the data class the method is part of. They are declared in the data class definition. In the POOSL data expression syntax, global variables look like this:

\[ E = x \]

This expression is used to read the instance variable within a data method. The counterpart in UML should be able to retrieve the *Property* of the *Class* from an *Activity*. An *Action* that is capable of doing this is *ReadStructuralFeatureAction*.

The *ReadStructuralFeatureAction* reads the value of a specific *StructuralFeature* like *Property*. It needs to know what class it is part of and thus it has an *InputPin*, called *object*, accepting the instance of the class. Since the global variable is always an instance variable of the present data class, the present class should be given to this *InputPin*. This is done with the *ReadSelfAction* which is explained in Section C.7.
The **ReadStructuralFeatureAction** returns the value of the specified global variable with the **OutputPin**, called **result**.

Figure C.2 shows **ReadStructuralFeatureAction** with the required metaclasses in the UML metamodel. The **OutputPin** enclosed by the ellipse has the value of the retrieved **Property**, in POOSL called the global variable.

An example is shown in Figure C.3. The names of the example in Section A.4 are used. To reduce the number of actions, the actions used for reading the instance variables can be replaced by an OCL expression, which will be assigned to a **ValuePin**. This **ValuePin** can be attached to the action that needs the instance variable. This OCL expression will be:

\[
\text{self.context.<instance_variable_name>}
\]

Via the **context** association, the instance of the **Class** that is the parent of the **Operation** can be accessed. The name of the instance variable can be used to retrieve the **Property**, which has the same name.

### C.3 Local Variable

Local variables look like this in the POOSL data expression syntax:

\[
E = u
\]

Readable POOSL local variables can be either parameters or locally defined variables. The only difference between the two in POOSL is that parameters are given from outside the scope of the method and local variables are in the scope of the method. Since the two will be mapped totally differently in UML, we will describe them separately.
APPENDIX C. DATA EXPRESSIONS

C.3.1 Parameter

POOSL methods can have parameters. The parameters are declared in the data method definition. The counterpart in UML must be able to retrieve the Parameter from the Activity. An ActivityNode that meets the requirement is ActivityParameterNode. ActivityParameterNode has a property parameter, which refers to the Parameter it should represent. Since the ActivityParameterNode represents an input parameter in this case, it can only have outgoing ObjectFlows.

Figure C.4 shows the UML metaclasses needed for modeling incoming parameters. The property direction of Parameter, which is not visible in the figure, has the value in.

Figure C.5 shows an example of incoming parameters using ActivityParameterNode.

C.3.2 Locally defined variable

In POOSL it is possible to declare variables locally. They are declared in the data method definition. In UML however, local variables are part of the Activity. There are two ways to do this.
C.3. LOCAL VARIABLE

To read the value of a local variable the action `ReadVariableAction` can be used. The property `variable` of the action needs to refer to a `Variable`, which is part of the `Activity`. The value of this `Variable` will be put on an `OutputPin`.

Figure C.6 shows the required elements in the UML metamodel. Figure C.7 gives an example of how local variables can be read. In the example the local variable `local` is being read.

**Property alternative**

Since an `Activity` is a `Class` it may have the property `ownedAttribute`, which is a list of relations between `Class` and `Property`. Reading `Properties` has been demonstrated in Section C.2. But there is a problem with doing the same as in Section C.2. The `ReadStructuralFeatureAction` requires a reference to the `Class`, with which the `Property` is associated. `ReadSelfAction` reads the class the method is part of and not the `Activity` where the modeled local variable (`Property`) is part of. No reference can be made while modeling this option in UML Activities. It can be represented by the given OCL expression, but there is no graphical way of representing it.

**C.3.3 OCL expression**

To read the instance values, it is possible to put an `InputPin` on the action that needs the value and enter an OCL expression to refer to the instance variable. Both modeling possibilities have the same OCL expression:
APPENDIX C. DATA EXPRESSIONS

Figure C.7: Example of reading a local variable in a UML activity diagram using the Variable alternative.

```
self.<local_variable_name>
```

The keyword `self` refers to the `Activity`. Since the metaclass `Property` and `Variable` are connected to `Activity`, both options can be accessed by this reference.

C.4 Literal

POOSL literals are expressions that represent a constant value. They cannot be instantiated. Examples are the number "3.14" and the string "Hello world". In the POOSL data expression syntax, literals look like this:

```
E = \gamma
```

In UML the passing of objects is done by `ObjectFlows`. These flows often flow into `InputPins`, which supply the input of an action. Literals are not instantiations of types however and do not need `ObjectFlows`. The `InputPin` remains, but in a specialized form, namely a `ValuePin`. Instead of the value being received from an `ObjectFlow`, the `ValuePin` has a property called `value`, which holds a `ValueSpecification`. The `ValueSpecification` can be specialized as an `OpaqueExpression`, which can hold the literal.

An alternative is that literals are passed on by a `ValueSpecificationAction`. This is an action, which holds a `ValueSpecification`, which should be used as described above. This action also has an `OutputPin`, on which this `ValueSpecification` is put.

Figure C.8 shows what UML metamodel elements are needed to map the POOSL constant expressions to UML. The `OpaqueExpression` holds in its body an OCL expression with the value of the `ValuePin`.

When representing a constant expression, it will look exactly like an `InputPin`. The only difference is that there are no incoming `ObjectFlows`, but a label with the constant expression in OCL is shown near the pin.

C.5 Return

POOSL data methods always have a resulting value. The keyword `return` returns the expression as the result of the data method.

In the POOSL data expression syntax, it looks like this:

```
E = return E
```
The returned expression should be of the same type as the return type of the data method. Like shown in Section B.2, in UML the result is modeled as a Parameter. How to map the regular POOSL parameters is shown in Section C.3.1. Mapping the returning parameter is similar. The only difference is that the ActivityParameterNode can only have incoming ObjectFlows instead of only outgoing ObjectFlows.

In POOSL it is possible to omit the keyword return and even to omit the line that indicates what should be returned. When no variable is explicitly returned, the method could return the object itself, unless the result is of another type. Then returning the POOSL data class Nil seems to be the best alternative.

Figure C.4 shows the elements needed in the UML metamodel. The only difference with incoming parameters is, that outgoing parameters have the direction set to return, see Section C.3.1. Figure C.9 gives an example of a value being returned in a UML activity.

C.6 Object creation

In POOSL data expressions the keyword new instantiates a class. In the POOSL data expression syntax, it looks like this:

\[ E = \text{new}(C) \]

where \( C \) is the type of the instantiated class.
APPENDIX C. DATA EXPRESSIONS

Figure C.10: CreateObjectAction in the UML metamodel

Figure C.11: Example class instantiation in a UML activity

UML has an action meant for object creation, called CreateObjectAction. It has a property classifier, which refers to the type of class that is instantiated. The instantiated class is put on an OutputPin.

Figure C.10 shows what elements in the UML metamodel are used to map the POOSL new command to UML.

In Figure C.11 an example is given how to show class instantiations in a UML activity diagram.

C.7 Self reference

When using the keyword self in POOSL, the instance of the class of the current method is being referred to. In the POOSL data expression syntax, it looks like this:

\[ E = \text{self} \]

In UML there is an action called ReadSelfAction, which reads the class of the current method. This action puts the class on an OutputPin.

Figure C.12 shows what elements are required in the UML metamodel to model the reference to the class in UML activity diagrams. An Activity can have a ReadSelfAction as a node. To reach the Class it refers to, it makes use of the context association between Activity and Class. This Class is accessible via the OutputPin.
C.8. DATA METHOD CALLS

As shown in Figure C.3, ReadSelfAction with its OutputPin is connected with an ObjectFlow to an InputPin. For designers it is cumbersome to create these shapes for a self reference. An alternative would be to use a ValuePin which owns an OCL expression (described in Section C.1). This OCL expression will look like this:

```plaintext
self.context
```

Via the `context` association, which is present in Figure C.12, the Activity can read the instance of the Class.

### C.8 Data Method Calls

In POOSL data methods it is possible to call other methods. This can be done with instantiated classes in the current scope, including the class where the current method is part of. In the POOSL data expression syntax, it looks like this:

\[
E = E(m(E_1, \ldots, E_n)) \\
E = E_{mC}(E_1, \ldots, E_n)
\]

There is a difference between dynamic and static method calls. The class, where the method is part of, can override methods of superclasses. The dynamic method calls call the overridden method and static method calls call the original method of the superclass.

To call methods in a UML Activity there is an Action called CallOperationAction. It needs the instantiated class (which can be retrieved with the action ReadSelfAction in Section C.7), at which the method to be called is a part of, and the parameters as input. These are only the parameters with option `direction = in`. Since parameters could have the same type they could be mixed in such a way that it is not visible what input belongs to what parameter. To overcome this problem all parameters lists will be ordered. This is not visible in the diagrams, but it is a property of the Parameter itself.

Figure C.13 shows what elements are used in the UML metamodel to map POOSL method calls to UML. There is a difference with the UML specification, CallOperationAction can only have one result OutputPin. The InputPin with the label `argument` takes the parameters, so the number and types are the same as the parameters of the Operation. The InputPin with label `target` refers to the data class, which method is called. The OutputPin with the label `result` is the parameter of the operation with the `direction` equal to return.
Figure C.13: CallOperationAction in the UML metamodel

Figure C.14: Example of method calls in a UML activity diagram

Figure C.14 shows an example of calling methods in a UML activity diagram. This example makes use of the TestClass class and the method setBuffer from Section A.4. The operation setBuffer is called from the present class itself (ReadSelfAction). A number is passed through an InputPin and the result flows out through an OutputPin.

To model static method calls it is possible to change the output type of the ReadSelfAction. Instead of the reference to the type of the present class, the OutputPin should refer to the type of the superclass. This way the operation of the superclass will be called instead of the overridden one.

C.9 Assignment to global variable

In POOSL, new values can be assigned to global variables. In POOSL data expressions, it looks like this:

\[ E = x := E \]

Section C.2 explains more about global variables.

Like mentioned earlier (in Section C.2), global variables are mapped into UML as a Property of a Class. To be able to assign a value to the Property an action is needed that can write to StructuralFeatures, like a Property. The action that can do this is the AddStructuralFeatureValueAction.
C.10 Assignment to Local Variable

In POOSL, new values can be assigned to local variables. In POOSL data expressions, it looks like this:

\[ E = u := E \]
In POOSL, new values can be assigned to local variables. Although Section C.3 explains that parameters are local variables, this is not the case for the assignment. So only locally defined variables will be described in this section.

Like described in the mapping of Section C.3, there are two ways to map locally defined variables into UML.

C.10.1 Variable

To assign a value to a local variable the action \textit{AddVariableValueAction} is used. The action requires the \textit{Variable} to be assigned in the property \textit{variable} and an \textit{InputPin} with the value that has to be assigned to the \textit{Variable}.

Figure C.17 shows the mapped metaclasses in the UML metamodel.

Figure C.18 gives an example of assigning a value to the local variable \textit{local}.

C.10.2 Property

The same problem holds as in Subsection C.3.2. In this case there is not even an OCL alternative.

C.11 Sequential composition

The complete expressions in POOSL are executed in a sequential order. Every two complete expressions are separated by a semicolon. In POOSL data expressions, it looks like this:
C.12. IF

POOSL has a conditional statement. It is implemented as an *if* statement. In POOSL data expressions, it looks like this:

\[ E = \text{if } E_c \text{ then } E_1 \text{ fi} \]
\[ E = \text{if } E_c \text{ then } E_1 \text{ else } E_2 \text{ fi} \]

There are two versions of the if statement. When the condition \( E_c \) is true, \( E_1 \) will be executed. When the condition \( E_c \) is false, \( E_2 \) will be executed if present.

In a UML activity, conditions are represented by a *DecisionNode*. To reproduce the POOSL behavior this *DecisionNode* will always have two outgoing edges (see Figure C.21 and C.22). The two paths are later on merged to one, using a *MergeNode*. The difference between the *if-then* and *if-then-else* option is that when the condition \( E_c \) is false, the former *if* statement terminates and the latter executes the *else* part (\( E_2 \)) and then terminates. The guards of the *ControlFlows* are written in OCL.

Figure C.20 shows what elements are required to model a POOSL *if* statement in UML.

In a mapping from UML to POOSL we should note that the only difference with the UML specification is that every *DecisionNode* has exactly two outgoing *ControlFlows* and every *MergeNode* has exactly two incoming *ControlFlows*.

To show the way *if-then-else* statement should be implemented two examples are given. In Figure C.21 a UML activity diagram is given with an *if-then* statement and in Figure C.22 an example of an *if-then-else* statement is given.
Figure C.20: Relation between DecisionNode and MergeNode in the UML metamodel

Figure C.21: Example of a if-then statement in a UML activity

Figure C.22: Example of a if-then-else statement in a UML activity
C.13 While

POOSL has a while statement. In POOSL data expressions, it looks like this:

\[ E = \text{while} \ E_c \text{ do } E \text{ od} \]

As long as \( E_c \) holds \( E \) will be executed. When at some point \( E_c \) evaluates to false, the statement ends.

Loops like POOSL while-do statements can be modeled in UML with the LoopNode. A LoopNode consists of three major parts. First there is a setupPart, which shows the initialization of the loop. Since POOSL while-do loops do not have this it will be omitted in the models. Then there is a test, which holds the condition. For the sake of consistency the condition is represented by a ValueSpecification (which is a ValueSpecificationAction), just like with if-then-else statements (in Section C.12). The last part that a LoopNode holds is the bodyPart, which holds a list of ExecutableNodes representing the body. Note that these can be specified to specific Actions.

Figure C.23 shows what elements are used in the UML metamodel to model while-do statements in UML. Like already mentioned, the setupPart is omitted. Since the condition is modeled as a ValueSpecificationAction, it will put its value, which should be a Boolean, on an OutputPin, which is read by the LoopNode. The ValueSpecificationAction also holds the condition in ValueSpecification, which is implemented as an OCL expression.

Figure C.24 shows how POOSL while-do statements can be shown in UML activity diagrams. Note that the OutputPin of Figure C.23 is not visible.
Figure C.24: Example of a while statement in a UML activity
Appendix D

Validation

In this appendix we validate the built transformation of this project, by transforming the modeled class Dictionary. The result and the calculation time is shown per transformation step.

The UML models are shown as a tree only. The Eclipse tools that support visual representation of UML models are either not complete or a commercial product. The tree view shows the relation between elements in the UML model.

The calculation time is the time the transformation took to transform one model into another model. The measuring of the calculation time is done on a laptop with a 1.86 GHz processor and 1 GB of RAM. During the test run only two Eclipse applications are running.

D.1 The starting UML model

Figure D.1 shows the tree of the method at of the class Dictionary. The UML model only includes the class and method headers of the class Dictionary and several class used by Dictionary, which together sum up to a total of 8 classes. Only the at method of Dictionary is implemented.

D.2 Step 1

The first transformation step of the implementation handles the grouping step without grouping conditional expressions. Figure D.2 shows the target model. This step took 2554 ms.

D.3 Step 2

The second transformation step of the implementation handles the grouping conditional expressions and sorting the structured activity nodes. These two steps are integrated for efficiency. When the conditional expression is built, the ordering of the branches is immediately correct. Figure D.3 shows the target model. This step took 2093 ms.

D.4 Step 3

The final step transforms the UML model into a POOSL model that also shown in tree view. The POOSL model is shown in Figure D.4 and the POOSL code is shown in Figure D.5. This step took 4306 ms.
Figure D.1: The starting UML model
Figure D.2: The UML model after the first transformation step
Figure D.3: The UML model after the second transformation step
Figure D.4: The POOSL model after the third transformation step
Figure D.5: The POOSL code after the third transformation step