From UML diagrams to behavioural source code

Sabrina L. Jim
Universiteit van Amsterdam

September 7, 2006

Centrum voor Wiskunde en Informatica

Thesis and internship supervisor :

prof. dr. P. Klint

Availability: public domain

Contents

Summary

Code generation from UML diagrams gains much attention lately in software engineering, because it has many benefits including the effort that is put into modelling is (fully) reused and generation of code is less error prone than writing code manually.

A framework that uses code generation is Model-Driven Architecture (MDA). The aim of MDA is to preserve the IT investments of companies despite the constant and rapid evolutions of platforms. Models in MDA are the key artefacts in all phases of development and are mostly expressed with Unified Modelling Language (UML). The latter is represented in XML Metadata Interchange (XMI).

MDA has a three abstraction layer approach, namely Platform Independent Model (PIM), Platform Specific Model (PSM), and code level. Functionality and behaviour of a system is saved in a PIM. After modelling, the PIM is transformed into one or more PSM automatically. Source code is generated from these PSMs.

Generation of behavioural source code can be done by means of UML behaviour diagrams. However, UML tools do not support the generation of source code from these diagrams, because (1) there is no one to one mapping between behaviour descriptions and the source code, (2) behaviour does not always have to appear in source code as explicit statements, (3) behaviour diagram can be implemented in different ways, and (4) the mapping between structure and behaviour diagrams is not always clear.

Due to all these problems, tools only generate the structure of the modelled system. MDA allows engineers to add source code to the generated code. The drawbacks of modifying the generated code are that the manually added code will be lost when engineers regenerate the source code from the models and errors can be introduced.

The goal in this project is to generate source code that also contains the structural as well as the behavioural aspect. As a result, modification to the generated source code is not needed. Note that we focus on the lowest generation step of MDA, which is from PSM to source code. The research question of this project is as follows:

How can behavioural code be generated using both structure and behaviour diagrams, despite the fact that UML does not provide a clear mapping between these types of diagrams?

In order to answer this question, we performed the following steps. (1) We selected a few UML diagrams and a UML tool. (2) We have built a test case system in order to verify the generated code in the end. (3) We investigated how we could extend/limit the modelling principles of UML. (4) We have built a generator that gives a tree format as output, which is used as input by a template-based generator (developed by Arnoldus). (5) The template-based generator generates source code, which we verify with our test case system (developed in step 2).

While we were executing these steps, we selected the state and the activity diagrams to model the behaviour and the class diagram to define the structure. Both behaviour diagrams can be modelled at the attribute level, which makes the gap between model and source code smaller. We concluded that these diagrams contain enough information for the generation of behavioural source code. The diagrams are modelled in a UML tool, called Poseidon.

While we were investigating the XMI that is generated by Poseidon, we discovered that XMI is very tool-specific, it contains references to other documents which are generated by other UML tools, it is very verbose, and it can expand to megabytes that lead to memory errors. That is why we transformed the XMI into another format (Interformat), which we defined by ourselves.

We use a UML profile to extend/limit UML that contains stereotypes and restrictions. While the stereotypes give us the possibility to define a behaviour in a concise way, the restrictions eliminate ambiguous interpretations of the behaviour diagrams.

In order to make a clear mapping between the structure diagram and the selected behaviour diagrams, we defined a rule in our UML profile that is defined as follows: *the diagram must be saved at the same level and have the same name as the related UML element.* As a result, the modelled behaviour will be generated at the right place in the source code.

Modifications in the generated source code are not needed any more, because it contains structure as well as behaviour aspects. As a result the source code is well-formed, it has the same meaning as the modelled behaviour, and it has the same architecture as code that is developed manually.

Preface

Although we are almost always working with computers, it is always about people. We create programs to simplify our jobs or make work process quicker. We try to use several techniques to make better programs. However, much programs that we make nowadays still contain errors. We are not able to remain the quality in each program that we make.

Many scientists are developing new techniques to ensure the quality of software. Sometimes a technique seems to be a real promise. As a consequence, many companies and engineers are running after it without thinking whether it is a real promise or not.

We think that new techniques do not always bring something new. Sometimes it is actually an old technique that is given a new face-lift with new buzz words. However, old techniques in another context can give new insights in old problems. We should investigate these new techniques carefully. Model Driven Architecture (MDA) is an example of a popular 'new' technique.

This thesis is the result of a master project of the one year Master program Software Engineering at the University of Amsterdam. This master project is about code generation from diagrams without the need to modify the generated source code.

The master project took us at least four months work. We tried to combine the techniques that we have learned during the lectures and practice. Many lessons are learned in this short time, including the following:

'It does not matter how slowly you go so long as you do not stop.'
'Learning without thought is labour lost; thought without learning is perilous.' - Confucius -

Sabrina Jim

Acknowledgement

First of all, I want to thank everybody who helped me somehow or other during this project, but are not mentioned below.

I want to thank Paul Klint for giving me the possibility to do my master project at the CWI and the supervision during this project. Jurgen Vinju and Mark van den Brand for the inspirations and discussions including transformations. Machiel Bruntink for helping me when I practice my biggest feature in ASF+SDF of creating cycles. My roommates for the fun, jokes, and support.

Bas, I want to thank you for everything, because you know more than anybody else what situations I had to face. I also want to thank my family: father, mother, and my sisters. I have learned how to deal with stress because of you.

Mrs. De Jong (Dutch): graag wil ik u bedanken voor alle steun die u mij gaf. Ook voor alle gesprekken die over van alles gaan. Het maakt het werk in het weekend leuker en lichter.

Chapter 1

Problem Description

Introduction

It is always about quality in the software industry. We are looking for ways to improve quality. As a consequence the development productivity decreases, because we have to take more time to check the source code that we wrote. As a result we are looking for ways to develop techniques in order to improve quality as well as increase development productivity. The effort that has been invested by engineers resulted in compilers, parsers, high-level languages, domain-specific languages, etcetera.

Engineers use various techniques to control the quality. An example of a technique is UML (Unified Modelling Language). UML is a general-purpose modelling language and is seen as the de facto standard. Engineers make UML diagrams of complex systems. A diagram is a simplification of the reality. Because of the simplification, engineers can understand systems that they are developing. The assumption is that better understanding leads to a system with higher quality.

Transformation and generation as a part of the development process

A known technique which focuses on the behaviour of a system is Model-Driven Architecture that is abbreviated as 'MDA'. Models in MDA are the key artefacts in all phases of development. The aim of MDA is to preserve the IT investments of companies despite the constant and rapid evolutions of platforms. Note that we use *diagram* and *model* interchangeably.

Functionality and behaviour of a system is saved in a Platform Independent Model (PIM). This model should be transformed in one or more Platform Specific Models (PSM) automatically. The models are mostly expressed with UML and are saved in a format that is based on XML (eXtensible Markup Language), called XMI (XML Metadata Interchange).

XMI is an OMG (Object Management Group) standard that is particularly developed to save UML diagrams in XML format. In spite of the OMG standard, many UML tools save the supported diagrams in their own XML-based format. The XML-based format contains data like classes, source code documentation, diagram lay-out, and tool-specific settings. As a consequence transformers and generators also have to use a tool-specific XMI as input for their transformation or generation process.

It is very difficult to generate behavioural source code automatically, because there are many ways to express a behaviour in a behaviour diagram. As a result, developing a generator that supports all possibilities is not practical to implement. Note that if we mention *behavioural source code* we mean source code that needs no further modification or extension, is well-formed, and can be compiled by a compiler. The compiled source code may contain run-time errors, but we will try to generate source code that contains no errors.

Although MDA promises a lot, it does not describe which diagrams engineers need to use in order to model functionality and behaviour. Of course, it depends on a specific situation which diagrams an engineer needs. However, we think that there is a subset of diagrams which can be applied to most systems, because some aspects are common. For example, every system has actions. As a result the activity diagram is always applicable.

Tools for generating source code

Many UML tools and all MDA tools support code generation from UML diagrams. A few examples of those tools are OptimalJ (Compuware), Enterprise Architect (Sparx Systems), Poseidon (Gentleware), ArgoUML (open source[1]), and MagicDraw (No Magic).

However, most of them are not able to generate behavioural source code, because they only generate source code from a class diagram. That is why those tools also allow engineers to modify the generated code. Source code which is added by hand will be lost if those engineers generate the source code again, or they have to do a lot of copying and pasting, which is very error prone.

Only a few tools like Fujaba[2] and MetaEdit[3] [LKT04] succeed in generating behavioural source code that is also executable. Fujaba uses UML in combination with a domain-specific language for embedded real-time systems. MetaEdit gain expressiveness by means of metamodelling (defining a new domain-specific language). As a result, engineers do not have to modify the generated code.

1.1 The problem

The questions that rise are 'Why do UML or MDA tools not generate behavioural source code? We already can generate static code from a class diagram. So, why can we not generate semantics from UML behaviour diagrams also?' We cannot give a precise answer to these questions, but we think that the following considerations help to get some insight in this matter.

A class diagram is easy to understand for an engineer. It gives an overview of classes, which have relations with each other. There is a one to one mapping between the diagram (modelled on the PSM level) and source code. However, a class diagram does not contain information about the behaviour of the system. As a consequence the generated source code does not contain behavioural source code either.

So, if engineers want to generate behavioural source code, they have to use diagrams that do contain behaviour information. UML has a set of diagrams to model the behaviour of a system: e.g. activity diagram, use case, state (machine) diagram, sequence diagram, communication diagram, interaction overview diagram, and timing diagram.

Unlike the class diagram, these types of diagrams are more difficult to draw. The reasons are as follows:

1. There is no one to one mapping between these behaviour diagrams and the source code (because behaviour can cover many source code statements, which appear at different places in a class or classes),
2. behaviour does not always have to appear in source code as explicit statements,
3. behaviour diagram can be implemented in different ways, and
4. the mapping between structure and behaviour diagrams is not always clear (which makes automatic code generation including behavioural aspects more difficult).

[1] ArgoUML - http://argouml.tigris.org/
[2] University of Paderborn, Software Engineering Group - http://www.fujaba.de/
[3] MetaCase - http://www.metacase.com/

For these reasons behaviour diagrams are not frequently supported for code generation in UML tools. As a result, behaviour diagrams are only used for communication purposes.

1.2 The research question

The aim of this project is to generate behavioural source code from UML diagrams. We formulated a research question and some subquestions in order to solve those problems.

The research question of this project is as follows:

'How can behavioural code be generated using both structure and behaviour diagrams, despite the fact that UML does not provide a clear mapping between these types of diagrams?'

In order to answer this research question we formulate some subquestions:

- *'How are behaviour diagrams saved in XMI, and is that representation practical for code generation?'*
  For code generation we need to know how behaviour diagrams are saved in the XMI format, because then we gain insight in the available data which we can use for the generation process. We will only investigate some selected behaviour diagrams (see Section 3.1.1 for the selected diagrams).

  This question does not have a clear mapping with the problems that are mentioned in the previous section.

- *'Which behaviour diagrams are suited for generating behavioural source code?'*
  We do not need to use all behaviour diagrams of UML to specify a particular behaviour. If we use all diagrams, the generation process will be very complex, because then the generator will have to take all diagrams into account and analyse them before it can generate source code. To keep the generation of behavioural source code simple, we will select a few diagrams.

  This question maps with the following problems: *there is no one to one mapping between behaviour descriptions and the source code* and *behaviour does not always have to appear in source code explicit as statements.*

- *'How can we constrain the modelling possibilities and simultaneously increase the expressiveness in behaviour diagrams?'*
  UML is a open and abstract language. Every diagram can be used independently and adding new 'artefacts' (elements that do not belong to the core set) is also possible. There are no strict modelling rules. Because of that, a generator must take all possibilities into account. For example, a `if-then-statement` can be modelled in at least three different ways in an activity diagram. By constraining the way of modelling, the generator does not have to take the exceptional situations into account. As a result, we can generate source code in a much more effective way. It also makes the behaviour diagrams easier to understand and we try to make a one to one mapping with the source code.

  This question maps with the following problems: *there is no one to one mapping between behaviour descriptions and the source code*, *behaviour does not always have to appear in source code explicit as statements*, and *a behaviour diagram can be implemented in different ways.*

- *'How can we make the mapping between structure and behaviour diagrams clear?'*
  As mentioned, every diagram can be used independently. There are no rules about when a diagram should be used to describe a feature or an element. For example, it is possible to use all diagrams to describe a use case. Code generation in this situation is only possible if it is done manually. By constraining the diagrams a generator knows what next step it must take in order to produce source code.

  This question maps with the following problem: *'the mapping between structure and behaviour diagrams is not always clear'.*

The models that we make in this project are modelled at the level of PSM. We investigate the generation from PSM to source code and we will discuss the tool(s) that we have developed.

1.2.1 The benefits of solving the problem

Answering the research questions has several benefits.

First, the effort that is put into modelling at the PSM level is (fully) reused. The behaviour diagrams are used for communication purposes, but now they are is also used as input data in order to generate source code.

Second, fully automatic code generation is less error prone than the traditional way. The source code is generated in a consistent way that needs no further modification or extension. As a result, there will be no syntax errors or faults caused by copying and pasting.

Third, fully automatic code generation decreases development time. It is possible to buy or to develop an own generator. In case of the latter, developing a generator can take months. Although the generated code can be developed in the same development time as the generator itself, the pay off of using a generator is when engineers are reusing it for another system. Generating source code for another system is done in just a few seconds, without developing a new generator again.

Fourth, design documentation is always up to date, because every change to the system will also be made in the structure and behaviour diagrams.

Last, it creates a *single point of definition*.

1.3 Research plan

The steps that we will take in this project are shown in Figure 1.1.

First, we select a UML tool and (structure and behaviour) diagrams that we use for modelling. Second, we develop a Java system manually to validate the generated system in the end. Third, we extend or limit UML, because UML is not expressive enough. We also make a model of the Java system with the selected diagrams. Fourth, we make a generator that generates an input (a BAST) for a template-based generator. Fifth, the template-based generator takes the BAST and generates a system. Last, we validate the generated system with the one that is developed manually. This last step will be done by ourselves.

First step We will select a UML tool which supports structure as well as behaviour diagrams and saves its diagrams in XMI. We need a UML tool that supports behaviour diagrams, because those are used to express behavioural source code. All UML tools support at least the use of a class diagram, which is a structure diagram. Because of that, we also use that structure diagram in the generation process. As a result, we are able to relate a behaviour to a specific class or method.

Second step We develop a Java test case system manually, which we will use as a test case to validate the system that we will generate. We have formulated a few requirements to determine whether the code that we are going to generate is good or not. The requirements are defined as follows.

1. *The code must be well-formed.*
   Like manually developed code, the generated source code has to be well-formed, because otherwise we cannot compile it (to byte code).
2. *The generated code must be semantically the same as the modelled behaviour.*
   It is obvious that the modelled behaviour in the diagrams must also appear in the generated code,

because that is what we want to generate in this project. However, we do not mean that the syntax of the generated code has to be exactly the same as the manually developed code.

3. *The architecture of the generated code should be the same as the code that is developed manually.* By defining this requirement, we reuse the effort that has been given while defining the architecture. With architecture we mean the structure and relations of classes. As a result of this requirement, the defined behaviour will generated in its related class.

Third step We extend or limit UML, because it is not expressive enough for generating behavioural source code. There are known techniques that are frequently used for this activity, namely metamodelling and using a UML profile. In this step we will decide which technique we will use.

We also make diagrams of the Java system in this step. These diagrams will be used as input for our generator that we will develop in the next step. In the end, the generator generates a Java system with the same behaviour as modelled in the diagrams.

Fourth step We develop a generator that takes UML diagrams as input and generates Java code. While generating source code, the generator merges the behaviour diagrams into one structure diagram. When merging is finished, the generator generates a BAST (Balance Abstract Syntax Tree) as output. That format is used by a template-based generator.

Fifth step We use a template-based generator developed by Arnoldus [Arn06]. The generator takes the generated BAST of the previous step as input and generates Java source code.

Last step We compare the generated source code with the manually developed one, which behaviours should be the same. This step will be done manually, because only then we can say that the generation of behaviour source code has been successfull.

Note that step two is discussed in Section 3.8.

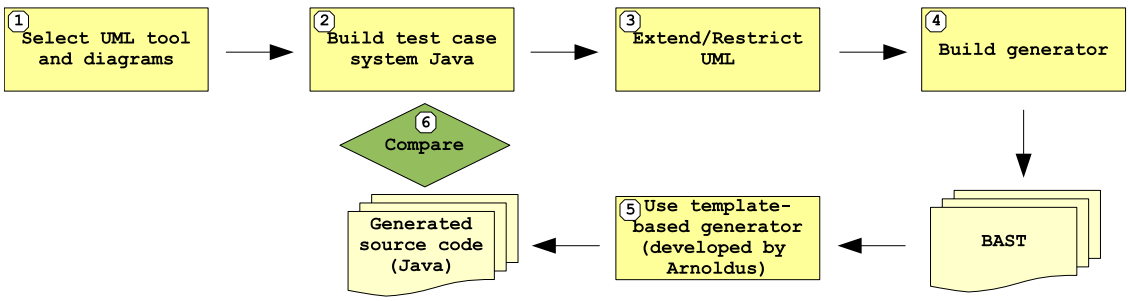


Figure 1.1: An abstract visualisation of plan of project

Chapter 2

Background and Context

Introduction

In this chapter we will discuss the techniques that we use. Those are Model Driven Architecture (MDA), Unified Modelling language (UML), XML Metadata Interchange (XMI), Extensible Stylesheet Language Transformations (XSLT), ASF+SDF, and Balance Abstract Syntax Tree (BAST).

2.1 Model Driven Architecture

The aim of MDA is to preserve the IT investments of companies despite the constant and rapid evolution of platforms (e.g. a specific programming language). MDA tries to capture investments in one Platform Independent Model (PIM) which should not be affected by major or minor changes in platforms. As the name of MDA already says, models are the key artefacts in all phases of development.

An MDA specification consists of a platform independent base UML model (PIM), plus one or more Platform Specific Models (PSM) and interface definition sets, each describing how the base model is implemented on a different platform. It focuses primarily on the functionality and behaviour of a distributed application or system (the PIM), not the technology in which it will be implemented. PSM takes the responsibility of describing the functionality and behaviour in one or more particular technologies.

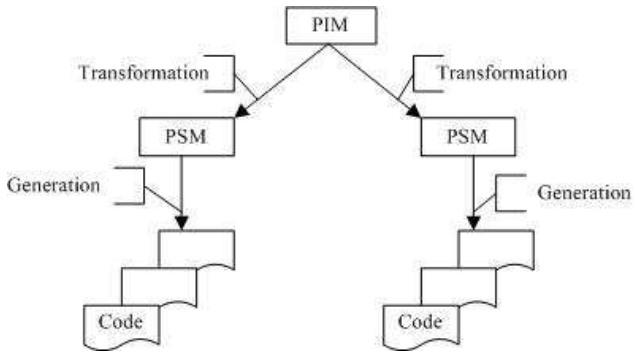


Figure 2.1: MDA hierarchy

By separating the functionality and behaviour from technologies, it is not necessary to model the system's functionality and behaviour again each time a new technology passes by. The PIM will be transformed by one or more transformers to PSMs. The lowest level of MDA is the source code level. The hierarchy of MDA

is shown in Figure 2.1.

To be able to transform a UML model from PIM to PSM (or from PSM to code), the model should contain enough information so that the transformation can be effective and complete. The transformer (or transformers) has to contain knowledge of the target platform and takes the PIM or PSM as input. Engineers have to change the transformer(s) and give the same PIM as input. This will result in one or more PSMs which use another platform than the previous PSMs.

2.2 Unified Modelling Language

Unified Modelling Language (UML) is a visual language in which engineers can model systems [FS00, WK01, OMG03, OMG04]. The objective of UML is to give engineers a formal and uniform way to communicate with each other without being ambiguous.

The UML was at first an attempt to unify various object-oriented modelling languages, and it seemed that its target applications were primarily business systems. It was and actually still is designed as a tool for communication between stakeholders. At this moment, UML is considered as the de facto standard for modelling and it is used for model transformation and code generation in MDA.

The UML models are expressed as diagrams which contain things (like classes, interfaces, and use cases) and relationships (like realisation, generalisation, and associations). There is no fixed number of diagrams in UML. However, most literature discusses only thirteen types of diagrams which are frequently used in practice. The thirteen diagrams are explained in [BRJ05].

The UML diagrams that we use in this project are the class diagram, the activity diagram, and the state diagram.

A class diagram is a view that shows a collection of classes, interfaces, and collaborations and their relationships. It is a structure diagram that does not show any behaviour elements, i.e. only static elements. It is seen as the most common diagram in modelling object-oriented systems.

An activity diagram is a behaviour diagram that shows the control flow and data from activity to activity across various objects.

A state diagram shows the control flow from state to state by means of states (in [BRJ05] mentioned as *state machine*), transitions, events, and activities. This diagram is seen as an important diagram for modelling the behaviour of an interface, class, or collaboration in [BRJ05, RFW+04].

For more information about how to use the diagrams we refer to [BRJ05, FS00, WK01].

UML tool: Poseidon for UML

We select a UML tool to work with in this project. The actual tool selection can be found in Section 3.1.2.

'Poseidon for UML' (frequently abbreviated as 'Poseidon') is a product of the company Gentleware[1]. The UML tool is based on the open source tool ArgoUML[2], which is licenced under the BSD licence[3]. The latter allows commercialisation of the open source product. Both projects are independent from each other.

The basic version of Gentleware's tool suite, Poseidon for UML Community Edition, is free of charge for non-commercial use. Poseidon for UML is more feature rich and more stable than ArgoUML.

For this project we use Poseidon for UML Community Edition version 4.1.0. The current downloadable version is 4.2. More information can be found on the website of Gentleware.

[1] http://www.gentleware.com/
[2] http://argouml.tigris.org/
[3] http://www.opensource.org/licenses/bsd-license.php

Level of precision

According to the UML user guide [BRJ05] diagrams can be used to model at different levels of precision (or abstraction). However, it does not distinguish which levels there are. It also does not explicitly describe whether the levels of precision are defined on the PIM level or on the PSM level. In this project we distinguish four different levels of precision on PSM level, namely the *architecture level*, the *class level*, the *method level*, and the *attribute level*.

Diagrams which are modelled at the architecture level include artefacts (like tables, files, and documents), nodes, components, and relations (like connection). The class level includes classes, attributes, methods, relations (dependencies, realisations). The method level includes attributes, messages, and events and timing (sequences and conditions of messages). The lowest level, attribute level, includes attributes, states, statements, and conditions of those statements. The lower the level of precision, the more expressiveness a diagram has.

Diagrams that we can use at the attribute level are state and activity diagrams. Examples of diagrams that are used on other levels are: sequence diagram at the method level, class diagram at the class level, and deployment diagram at the architecture level.

Extending UML

UML 2.0 is particularly developed to support the principles of modelling in MDA. It is very hard to transform a model to an executable system automatically, because of missing information. However, UML was never designed for transformation and generation of executable code.

UML is a modelling language which gives the possibility to solve a problem in an abstract way, as a result details are omitted. As a consequence, UML is not expressive enough. For code generation, UML misses expressiveness at the attribute level. For example, it is difficult to express exception handling in an activity diagram. Engineers try to compensate the lack of expressiveness by extending UML by means of metamodelling [Béz05] or making profiles [BRJ05, SW06].

Metamodelling is the activity of making a metamodel. A metamodel describes the various kinds of contained model elements, the way they are arranged, related, and constrained [Béz05].

An engineer can extend the metamodel of UML by adding new relation types and elements or by redefining them. It is also possible to make an own (domain-specific) modelling language by making a new metamodel instead. In that way, engineers are not limited to the metamodel of UML.

Extending UML by means of a UML profile means that the metamodels will not be modified. UML has a few parts that engineers may modify: stereotype, tagged values, constraints, and base classes. A *profile* is 'a set of predefined stereotype, tagged values, constraints, and base classes [BRJ05].' UML models that are modelled with a profile still conform to the original UML metamodel. A profile only 'annotates' the relations and elements with stereotypes, tags, or constrains the UML elements.

The constraints of a UML profile are conditions between elements that a run-time configuration must satisfy to conform to the model. It may be written with UML's Object Constraint Language (OCL) or as free-form text. The latter will not be interpretable for automated code generation, because it is unknown how to parse the text and what to generate. In case of OCL, the generator has a set of keywords that an engineer can use. Depending on the annotated UML elements and relations, the generator can generate behavioural source code.

2.3 XML Metadata Interchange

When the first tools for UML appeared, there was no standard format for interchange of UML models. Many UML tools had their own (textual) format. To increase the interchange of UML models between tools and

metadata (data that describes other data) repositories, OMG developed a format called XMI which is a abbreviation of XML Metadata Interchange [OMG02].

Although XMI indeed increased the interchange of UML models, there is still a variety of formats. Every UML tool (which uses XMI as its format to save UML models) has its own tool-specific XMI format. This is caused by inconsistency in the XMI specification, incompleteness of the specification, and it is difficult to implement the specification in UML tools.

The XMI is not developed to be human readable. So, engineers who see an XMI document for the first time can be discouraged to use it, because there are many different tags and attributes which are arranged in a huge tree structure. We found 142 different tags and 31 different attributes in our test case. An XMI document is very verbose which can easily expand to megabytes. Note that we will use *tags* and *XMI element* interchangeably.

The current version of XMI is 2.1 [OMG05]. The UML tool that we have selected in Section 3.1.2 saves its diagram in XMI version 1.2 [OMG02]. As a result of the latter, we use the older XMI version (1.2) in this project.

Specifications of XMI

OMG made a XMI specification in EBNF (Extended Backus-Naur Form) and DTD (Document Type Definition). We noticed that the EBNF specification is not very large (10 pages long) in comparison with the DTD specification (117 pages long).

The difference in size is caused by the level of abstraction. It is possible to define only the structure and the character class of an element name or attribute name, while DTD specifications are forced to define concrete element and attribute names.

We only discuss the EBNF specification of XMI, because we used this specification in this project. While we were investigating the XMI we discovered some interesting errors in the specifications. The results of this are discussed in Appendix A.

Structure of XMI

An XMI document contains a `header`, `content`, `differences`, and `extensions`, where `header` includes supplier and tool version information, `content` contains information of the models, `differences` contains the modifications of a XMI document, and `extensions` contains information which is unrestricted. In Listing 2.1 the main structure of an XMI document is shown.

The number of different tags depends on which UML tool is used to generate an XMI file (because, as mentioned, every tool has its own XMI format). Although there are many different tags, mostly those tags are straightforward and understandable like `<UML:Class ...>` and `<UML:Interface ...>`.

An example of a tag which is not straightforward is `<UML:ModelElement ...>`. Because everything is an element in UML, we thought that it can contain elements of classes, interfaces, dependencies, et cetera. However, this tag contains only information about stereotypes and tagged values.

There are two different kinds of tags in XMI, namely a tag that contains data and a tag that does not contain data. A tag that contains data is called `ObjectStart` in the specification, while the other tag that does not contain data is called `ItemHdr` (pronounced as *item header*) [OMG02].

XMI has a deep nesting structure which is caused by the hierarchy of elements in UML. In this project we use an XMI document as a source file example which is based on a small application. The XMI document has sixteen nested levels, contains more than 65,000 lines and is 4.1 megabytes in size.

Listing 2.1: An XMI document main structure

```
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML'
```

```
  timestamp = 'Tue May 16 14:05:05 CEST 2006'>
  <XMI.header>
    <!-- header content -->
  </XMI.header>
  <XMI.content>
    <!-- Model information -->
  </XMI.content>
  <XMI.difference>
    <!-- Modifications of a XMI document -->
  </XMI.difference>
  <XMI.extensions xmi.extender=...>
    <!-- Unrestricted content -->
  </XMI.extensions>
</XMI>
```

2.4 Mapping between UML behaviour diagrams and XMI

In this section we discuss the mapping between selected UML behaviour diagrams (activity and state diagram, see Section 3.1.1) and XMI that is generated by Gentleware's Poseidon UML tool. We will explain why we have chosen for this UML tool in Section 3.1.2.

As mentioned before, each UML tool generates its tool-specific XMI. As a consequence, the mapping that is discussed here is only applicable for the selected UML tool.

An XMI contains data of the diagrams, including layout of the diagrams. Because of the fact that there is no strict separation between real data (information that is needed for transformation) and layout, the latter can appear together in an XMI element (a tag with attributes) or apart. It can also appear in different forms.

While transforming an XMI it is important not to loose data that contains information of the source code that has to be generated. In order to decide whether an XMI element or attribute is important we have to gain insight in the XMI. That is also the reason why we discuss this step.

2.4.1 Mapping between activity diagram and XMI

We modelled several activity diagrams before we analysed the XMI. For each activity diagram we modelled the objects and the different relations (with and without condition) between them. We also placed them at several places in the tree of the UML project tree (see Figure 2.2), because we can see whether the the element names of the tags will change. We noticed that the position of an activity diagram in the UML project tree determines how the diagrams are indicated.

An activity diagram is separated in two parts, `Activity.edge` and `Activity.node`. The `Activity.edge` contains activity edges (relations between activities), while the `Activity.node` contains activity nodes (activities).

Each activity edge contains information about the activity which triggered the relation (`ActivityEdge.source`), the target activity (`ActivityEdge.target`). An activity edge can also have a guard (`ActivityEdge.guard`, a condition). The guard is expressed as text. Only when that condition is true, the relation will trigger the target activity.

An example of an activity edge without a guard is shown in Figure 2.3 and the corresponding XMI is shown in Listing 2.2. There are two XMI references to other call nodes (XMI elements). The names of the call nodes (`Action_3` and `Action_4`) are saved in the referred XMI elements (see Figure 2.7). Note that we will use *call node* and *call action* interchangeably, because Poseidon use these two terms as such.

Another example of an activity edge with a guard is shown in Figure 2.4 and the corresponding XMI is

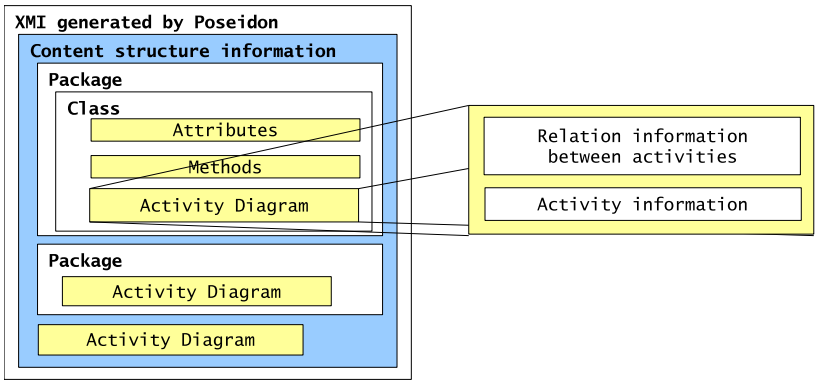
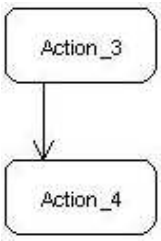


Figure 2.2: Position of activity diagram in project

Figure 2.3:

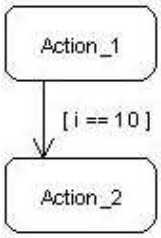


Listing 2.2: Mapping with XMI element, relation without guard

```
<UML2:ActivityEdge xmi.id = 'I1223bf0m10b3c37528dmm76e4'
  visibility = 'public' isSpecification = 'false'>
  <UML2:ActivityEdge.target>
    <UML2:CallNode xmi.idref = 'I1223bf0m10b3c37528dmm76e9'/>
  </UML2:ActivityEdge.target>
  <UML2:ActivityEdge.source>
    <UML2:CallNode xmi.idref = 'I1223bf0m10b3c37528dmm76fc'/>
  </UML2:ActivityEdge.source>
</UML2:ActivityEdge>
```

shown in Listing 2.3. Note that the XMI tag of the relation with a guard has extra attributes (`visibility`, `isSpecification`, `body`, and `language`). These attributes will always be generated by Poseidon, but it has no special meaning or added value to the relation.

Figure 2.4:



Listing 2.3: Mapping with XMI element, relation with guard

```
<UML2:ActivityEdge xmi.id = 'I1223bf0m10b3c37528dmm76cf' name = ''
  visibility = 'public' isSpecification = 'false'>
  <UML2:ActivityEdge.guard>
    <UML2:OpaqueExpression xmi.id = 'I1223bf0m10b3c37528dmm76a1'
      name = 'i == 10' visibility = 'public'
      isSpecification = 'false' body = 'true' language = 'java'/>
  </UML2:ActivityEdge.guard>
  <UML2:ActivityEdge.target>
    <UML2:CallNode xmi.idref = 'I1223bf0m10b3c37528dmm76f7'/>
  </UML2:ActivityEdge.target>
  <UML2:ActivityEdge.source>
    <UML2:CallNode xmi.idref = 'I1223bf0m10b3c37528dmm76e9'/>
  </UML2:ActivityEdge.source>
</UML2:ActivityEdge>
```

Each activity diagram starts with an initial node (see Figure 2.5) and ends with a final node (see Figure 2.6). An initial node has only one relation that goes to a UML element. A final node has one or more relations that refer to the node self. The mapping with XMI of the initial and the final node are given respectively in Listing 2.4 and Listing 2.5.

A call node is used to specify an activity (see Figure 2.7). In our case, a statement or an annotation. A call node can contain one or more incoming relations, while it has only one outgoing relation. The corresponding

Listing 2.4: Mapping with XMI element, initial node

Figure 2.5:

Initial Node

```
<UML2:InitialNode xmi.id = 'I1223bf0m10b3c37528dmm76fc'
  name = '' visibility = 'public'
  isSpecification = 'true'>
  <UML2:ActivityNode.outgoingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm76e4'/>
  </UML2:ActivityNode.outgoingEdge>
</UML2:InitialNode>
```

Listing 2.5: Mapping with XMI element, activity final node

Figure 2.6:

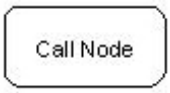
Activity Final Node

```
<UML2:ActivityFinalNode xmi.id = 'I1223bf0m10b3c37528dmm7690'
  name = '' visibility = 'public'
  isSpecification = 'true'>
  <UML2:ActivityNode.incomingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm768b'/>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm7676'/>
  </UML2:ActivityNode.incomingEdge>
</UML2:ActivityFinalNode>
```

XMI of a call node is shown in Listing 2.6.

Listing 2.6: Mapping with XMI element, call node

Figure 2.7:




```
<UML2:CallAction xmi.id = 'I1223bf0m10b3c37528dmm76f7'
  name = '' visibility = 'public'
  isSpecification = 'false'>
  <UML2:ActivityNode.incomingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm76cf'/>
  </UML2:ActivityNode.incomingEdge>
  <UML2:ActivityNode.outgoingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm768b'/>
  </UML2:ActivityNode.outgoingEdge>
</UML2:CallAction>
```

A decision node and merge node (see Figure 2.8 and Figure 2.9 respectively) are syntactically the same, both nodes have a shape of a diamond. While the decision node announce different paths of activities, a merge node merges or unifies different paths together. The corresponding XMI of a decision node is shown in Listing 2.7. The XMI representation of a merge node is shown in Listing 2.8.

Listing 2.7: Mapping with XMI element, decision node

Figure 2.8:

Decision Node

```
<UML2:DecisionNode xmi.id = 'I1223bf0m10b3c37528dmm76e9'
  name = '' visibility = 'public' isSpecification = 'true'>
  <UML2:ActivityNode.incomingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm76e4'/>
  </UML2:ActivityNode.incomingEdge>
  <UML2:ActivityNode.outgoingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm76cf'/>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm76ba'/>
  </UML2:ActivityNode.outgoingEdge>
</UML2:DecisionNode>
```

Listing 2.8: Mapping with XMI element, merge node

```
<UML2:MergeNode xmi.id = 'I1223bf0m10b3c37528dmm7762'
  name = '' visibility = 'public' isSpecification = 'true'>
  <UML2:ActivityNode.incomingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm77ce'/>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm775d'/>
  </UML2:ActivityNode.incomingEdge>
  <UML2:ActivityNode.outgoingEdge>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm77f0'/>
    <UML2:ActivityEdge xmi.idref = 'I1223bf0m10b3c37528dmm7723'/>
  </UML2:ActivityNode.outgoingEdge>
</UML2:MergeNode>
```

Figure 2.9:

2.4.2 Mapping between state diagram and XMI

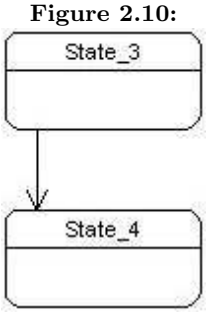
We performed the same steps as we did for the activity diagrams. We made state diagrams and placed them at several places in the tree of the UML project. For each state diagram we tried to model all possible combinations between the different (UML) state elements and relations. Just like the activity, the position of a state diagram in the UML project determines how the diagrams are announced.

A state diagram is separated in two parts, namely `Region.subvertex` and `Region.transition`. The `Region.subvertex` contains all states and substates information. The `Region.transition` contains transitions (relations between states) information.

Each relation (transition) contains information which triggered the transition (`Transition.source`) and the target state (`Transition.target`). Some relations also have a guard which is saved as `Transition.guard`. The guard is an expression which must be true in order to trigger the target state.

An example of a relation without a guard is shown in Figure 2.10 and its corresponding XMI is shown in Listing 2.9. An example of a relation with a guard is shown in Figure 2.11. The corresponding XMI is shown in Listing 2.10.

Figure 2.10:



Listing 2.9: Mapping with XMI element, relation without guard

```
<UML2:Transition xmi.id = 'I98a7ebm10cd42e77e6mm55a4'
  visibility = 'public' isSpecification = 'false'
  kind = 'external'>
  <UML2:Transition.source>
    <UML2:State xmi.idref = 'I98a7ebm10cd42e77e6mm55a9'/>
  </UML2:Transition.source>
  <UML2:Transition.target>
    <UML2:State xmi.idref = 'I98a7ebm10cd42e77e6mm55b4'/>
  </UML2:Transition.target>
</UML2:Transition>
```

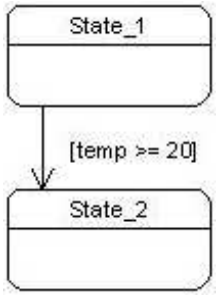
Like the activity diagram, the state diagram also has an initial and final node. However, in this diagram these nodes are known as initial state (see Figure 2.12) and final state (see Figure 2.13) respectively. The initial state has a outgoing relation to a state. It cannot have more than one relation to a particular state. The final state has one or more incoming relations from a particular state to itself. The corresponding XMI of an initial node is shown in Listing 2.11, while the XMI source code of a final state is shown in Listing 2.12.

A state is a condition or situation and can have one or more incoming and outgoing relations. It has the shape of a rounded rectangle (see Figure 2.14). The state `Simple State` that has one incoming and one outgoing relation is shown in Listing 2.13.

Listing 2.10: Mapping with XMI element, relation with guard

```
<UML2:Transition xmi.id = 'I98a7ebm10cd42e77e6mm54f0'
  visibility = 'public' isSpecification = 'false'
  kind = 'external'>
  <UML2:Transition.source>
    <UML2:State xmi.idref = 'I98a7ebm10cd42e77e6mm5658'/>
  </UML2:Transition.source>
  <UML2:Transition.target>
    <UML2:State xmi.idref = 'I98a7ebm10cd42e77e6mm55e0'/>
  </UML2:Transition.target>
  <UML2:Transition.guard>
    <UML2:Constraint xmi.id = 'I98a7ebm10cd42e77e6mm54db'
      name = '' visibility = 'public' isSpecification = 'false'>
      <UML2:Constraint.specification>
        <UML2:OpaqueExpression xmi.id = 'I98a7ebm10cd42e77e6mm54d4'
          name = '' visibility = 'public' isSpecification = 'false'
          body = 'temp >= 20' language = 'java'/>
      </UML2:Constraint.specification>
    </UML2:Constraint>
  </UML2:Transition.guard>
</UML2:Transition>
```

Figure 2.11:



Listing 2.11: Mapping with XMI element, initial state

```
<UML2:Pseudostate xmi.id = 'I734770m10c483e9153mm57a8'
  name = '' visibility = 'public' isSpecification = 'false'
  kind = 'initial'>
  <UML2:Vertex.outgoing>
    <UML2:Transition xmi.idref = 'I98a7ebm10cd42e77e6mm5642'/>
  </UML2:Vertex.outgoing>
</UML2:Pseudostate>
```

Figure 2.12:

Initial State

Listing 2.12: Mapping with XMI element, final state

```
<UML2:FinalState xmi.id = 'I19b60bbm10cd45e4e27mm7b2b'
  name = '' visibility = 'public' isSpecification = 'false'>
  <UML2:Vertex.incoming>
    <UML2:Transition xmi.idref = 'I98a7ebm10cd42e77e6mm565g'/>
  </UML2:Vertex.incoming>
</UML2:FinalState>
```

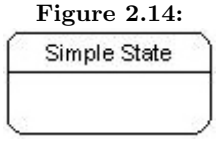
Figure 2.13:

Final State

Listing 2.13: Mapping with XMI element, simple state

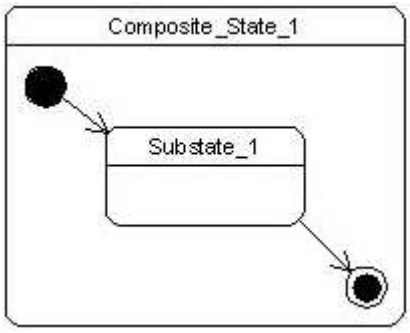
```
<UML2:State xmi.id = 'I98a7ebm10cd42e77e6mm564d'
  name = 'Simple State' visibility = 'public'
  isSpecification = 'false'>
  <UML2:Vertex.outgoing>
    <UML2:Transition xmi.idref = 'I98a7ebm10cd42e77e6mm55ce'/>
  </UML2:Vertex.outgoing>
  <UML2:Vertex.incoming>
    <UML2:Transition xmi.idref = 'I98a7ebm10cd42e77e6mm562d'/>
  </UML2:Vertex.incoming>
</UML2:State>
```

Figure 2.14:



The composite state is a state that contains other substates or nonorthogonal states. States that are not a part of the composite state can have a relation with the composite state or a substate in the composite state. A composite may have at most one initial and final state. An example of a composite state is shown in Figure 2.15 and is saved in the XMI format as shown in Listing 2.14. Note that the XMI only contains the structure of a composite. The substates are omitted and are referred as `<!-- Content of region -->`.

Figure 2.15:



Listing 2.14: Mapping with XMI element, composite state

```
<UML2:State xmi.id = 'I98a7ebm10cd42e77e6mm55e0'
  name = 'Composite_State_1' visibility = 'public'
  isSpecification = 'false'>
  <UML2:Vertex.incoming>
    <UML2:Transition
      xmi.idref = 'I98a7ebm10cd42e77e6mm54f0'/>
  </UML2:Vertex.incoming>
  <UML2:State.region>
    <UML2:Region xmi.id = 'I98a7ebm10cd42e77e6mm55df'
      name = 'Region_1' visibility = 'public'
      isSpecification = 'false'>
      <UML2:Region.subvertex>
        <!-- Content of region -->
      </UML2:Region.subvertex>
    </UML2:Region>
  </UML2:State.region>
</UML2:State>
```

The orthogonal state is similar to a composite state. The difference is that the orthogonal state have two or more state machines (regions) that execute in parallel in the context of the enclosing object. Each region has only one initial and final states. An example of an orthogonal state is shown in Figure 2.16. A mapping with the XMI is shown in Listing 2.15. Note that the content of two state machines are saved in `<!-- Content of region 1 -->` and in `<!-- Content of region 2 -->`.

A submachine is actually a reference to another state machine. A submachine is shown in Figure 2.17 and the corresponding XMI is shown in Listing 2.16. Note that the figure does not have a reference yet, but in the XMI source we can see it has a reference to another state diagram where `xmi.idref` has the value `I734770m10c483e9153mm7cf1`.

Note that there are different kinds of states (simple, composite, and orthogonal). Unlike activity diagram where every different element has another element name, all states are saved with the element name `State`. The only difference between the composite and the orthogonal state is that the orthogonal has more than one region that contains substates.

2.5 Implementation languages

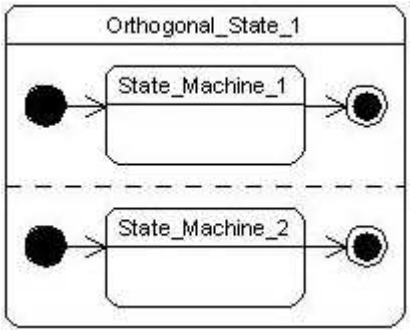
We have chosen for XSLT and ASF+SDF as potential implementation language for the transformers that we make (see Figure 1.1). Both techniques are designed for making transformations between different formats.

We evaluate for each transformer which technique is the best to use as implementation language. The evaluations are based on our understanding of the techniques as discussed below.

2.5.1 XSLT

XSLT is an abbreviation for eXtensible Stylesheet Language Transformations. It is developed by the W3C organisation especially for transformations of XML documents. XSLT is a core technology for processing

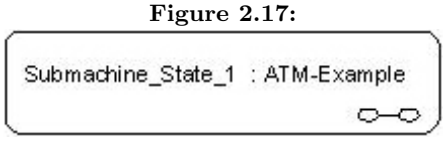
Figure 2.16:



Listing 2.15: Mapping with XMI element, orthogonal state

```
<UML2:State xmi.id = 'I1ebbfdem10cd477175amm7afd'
  name = 'Orthogonal_State_1' visibility = 'public'
  isSpecification = 'false'>
  <UML2:Vertex.outgoing>
    <UML2:Transition
      xmi.idref = 'I1ebbfdem10cd477175amm79a7'/>
  </UML2:Vertex.outgoing>
  <UML2:Vertex.incoming>
    <UML2:Transition
      xmi.idref = 'I1ebbfdem10cd477175amm79c8'/>
  </UML2:Vertex.incoming>
  <UML2:State.region>
    <UML2:Region xmi.id = 'I1ebbfdem10cd477175amm7afc'
      name = 'Region_1' visibility = 'public'
      isSpecification = 'false'>
      <!-- Content of region 1 -->
    </UML2:Region>
    <UML2:Region xmi.id = 'I1ebbfdem10cd477175amm7afb'
      name = 'Region_2' visibility = 'public'
      isSpecification = 'false'>
      <!-- Content of region 2 -->
    </UML2:Region>
  </UML2:State.region>
</UML2:State>
```

Figure 2.17:



Listing 2.16: Mapping with XMI element, submachine

```
<UML2:State xmi.id = 'I734770m10c483e9153mm7ce8'
  name = 'Submachine_State_1' visibility = 'public'
  isSpecification = 'false'>
  <UML2:State.submachine>
    <UML2:StateMachine
      xmi.idref = 'I734770m10c483e9153mm7cf1'/>
  </UML2:State.submachine>
</UML2:State>
```

XML.

XSLT is a general-purpose translation tool, a system for reorganising document content, and a way to generate multiple results from the same content. With XSLT we can add/remove elements and attributes to or from the output file. We can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, et cetera.

Using XSLT we can easily traverse through XML based documents and select particular data or transform a particular part of the tree into another presentation like HTML, webpages, WAP, and SVG. We only have to give the element name, which XSLT uses to match the elements and place the information in the holes of a template.

However, an element will only be transformed if the corresponding template is called. So, when developing a transformer in XSLT, the engineer has to know exactly where elements can appear, because it is very easy to miss some elements. This can lead to a transformed document where information is lost.

Because the XSLT processor has no knowledge of the target language, transformers which are implemented in this language cannot ensure that the target presentation has the correct syntax. However, if we transform the code into another XML based format, the XSLT processor will take care that the generated document does conform to the XML syntax.

XSLT does not have a stack machine that manages the creation and initialisation of the variables. All variables that are used can remember one value while processing only. If a variable is assigned a new value, the old one will be lost. As a consequence, recursive calls are not possible, if the called template contains variables.

The language has a limited set of functions, which we can use to modify characters. It is possible to extend the functions in XSLT, but then we need knowledge about another technology like (javascript) scripting language or C# (to extend the XSLT processor), or we use EXSLT (Extended XSLT[4]).

Because XSLT is standard used by engineers to transform XML, we want to use it as an implementation language for the transformers that we make. More about XSLT can be found in [ABC+01, Cla99]. A consequence of this is that we probably have to change the XSLT processor. Note that we use Apache's open source XSLT parser and processor.

2.5.2 ASF+SDF

The specification formalism ASF+SDF is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. It is a specification language where we can define the syntax and the semantics of programming languages. We can also specify rewrite rules for a compiler or a transformer, which transforms Java source code to byte code for example.

Unlike XSLT, we can transform any language into another as long as we have the grammars of the languages. As a result, the benefit of using ASF+SDF as implementation language is that we can ensure that the target presentation is correct. Although this also can be seen as a drawback, because we always have to have the grammar of the target language. For example, if we want to transform an XMI into a text file with out any format, we have to define a grammar for that text.

Functions are defined and saved in two different parts. The syntax of a function is defined in the SDF part, while the rewrite rules of that function are defined in the ASF part. So, if we have to modify a function, we probably have to make a change in the ASF part. Then we also have to check other functions that use that particular function.

Depending on the type of functions (whether those are traversals) that do the transformation, we sometimes need more rules to perform a transformation than in the case of XSLT. As a consequence, the implementation of a transformer can be very large.

[4] http://www.exslt.org/

While making a transformer in ASF+SDF, we do not have to worry about the values that are saved in created variables. The variables of the rewrite rules are saved on a stack. Each time a rewrite rule is applied, new variables are created. When a rewrite rule is finished, old variables will be thrown away.

In ASF+SDF there are no predefined functions to modify characters of a lexical sort, except for the lexical constructor functions. Functions have to be created for every modification that is needed. The benefit is that it results in a set of functions that is very expressive. The drawback is that it takes the engineer more time to develop a transformer.

We want to use ASF+SDF as an potentional implementation language, because the weakness of XSLT is the strength of ASF+SDF and vice versa. Another reason is because we already are acquainted with this language. Literature about ASF+SDF can be found in [BK05].

2.6 Template-based generator

The template-based generator described in [Arn06] takes a BAST (see Section 2.6.1) and templates as input in order to generate source code. The generator has no name yet, so we will refer to it as 'template-based generator'.

The generator is similar to the processor of XSLT. Like the XSLT processor, it also traverse through a tree in a fixed format by means of a given selection query. The query language, which the generator uses, has the same control constructs as XSLT. The biggest difference between the XSLT processor and the generator is that the latter generates syntactically correct code.

Before the generation process, the generator parses the given template(s) and the BAST. A template contains holes, which are instructions for the generator to traverse through the tree. In this way, the generator ensures that the generated source code is correct. A notable thing is that the generator can generate more than just one language, like Java, C++, C, et cetera [Arn06].

The generator is implemented in ASF+SDF.

Why we use this generator

We could make a generator by ourselves, which transforms our XML-based document into Java. To make sure that the generated Java code is also syntactically correct, we could implement the generator in ASF+SDF. In the end we would get a generator, which is functionality actually the same as the template-based generator.

The difference between our implementation and the template-based generator is that our generator would only generate Java code, because that is what we want to generate.

If we use the template-based generator, we still have to make a transformer that transforms the interformat into a BAST. Unlike the Java grammar, the BAST grammar is very small. This will save us much time in order to become acquainted with it.

2.6.1 Balance Abstract Syntax Tree

The BAST (Balance Abstract Syntax Tree) is a format that is used as input for a template-based generator. As the name of BAST already suggest, the format has a structure of a tree and it starts with the keyword `bAST` (see Listing 2.17) [Arn06].

Listing 2.17: SDF definition of BAST, rule one

```
"bAST" "(" "[" {BRecord ","}* "]" ")" -> BAST
```

The tree consist of a list with four different sorts, namely a `bDomain`, `bLookup`, `bClass`, and a `bAttribute`. We need to use the last two sorts in order to define a class and its attributes. Note that the first two sorts are used to represent data from the database domain. The syntax definition of the sorts are shown in Listing 2.18.

Listing 2.18: SDF definition of BAST, the four different sorts

```
BProperty | BClass | BLookup | BDomain          -> BRecord

"bDomain" "(" "[" { BProperty "," }+ "]" ")"    -> BDomain
"bLookup" "(" "[" { BProperty "," }+ "]" ")"    -> BLookup
"bClass" "(" "[" { BClassItem "," }+ "]" ")"    -> BClass
"bAttribute" "(" "[" { BProperty ","}* "]" ")"  -> BAttribute

BProperty | BAttribute                          -> BClassItem
```

A `bProperty` is indicated with a given custom name (`UQLiteral "("BElem ")"`) that can contain a primitive sort. The four sorts that are possible are `real`, `int`, `bool`, and `string`. We have shown the syntax of these sorts in Listing 2.19.

Listing 2.19: SDF definition of BAST, the four primitive sorts

```
UQLiteral "(" BElem ")"         -> BProperty {cons("belem")}
"bool" "(" real-value:BoolCon ")"       -> BElem {cons("bool")}
"real" "(" real-value:RealCon ")"       -> BElem {cons("real")}
"int" "(" int-value:IntCon ")"  -> BElem {cons("int")}
"str" "(" str-value:StrCon ")"  -> BElem {cons("str")}
```

The syntax of the BAST gives us the possibility to define a part of the tree. However, because we do not generate source code where attributes are the main issue. We also need a sort that we can use to define our Java operations. This issue will be discussed in Section 3.6.

The difference between BAST and XMI

There are two differences between the BAST format and the XMI format. The first difference is that a BAST does not have references to other elements in the tree, while an XMI does. An XMI can also have references to elements in other documents or even a whole document. This makes a BAST very verbose.

The second difference is that elements in a BAST has only a small set of sorts to indicate the data, while an XMI has many (142) different elements.

2.6.2 Templates for the template-based generator

Like every template, this templates contains holes, which will be used by the generator to fill it up with data. The holes are indicated with `<% expression %>`. The expression that is defined in the holes matches with a given custom name or a sort.

The expressions are similar to the expressions in XSLT. The difference is that it is possible to define small reusable templates in XSLT, while the expression is a part of a template in our case. If we want to reuse the same data, we have to define the expression again.

Fortunately, it is possible to use `<%for each ... do%>` to repeat the same defined action. An example of a template for generating classes is shown in Listing 2.20.

Listing 2.20: A template that is used by the template-based generator

```
<%for each bClass in bAST do
```

```
  //Save each class in a file as follows <class name>.java
  store in <%.bAST.name%>/<%name || ".java"%>
%>
package <%containedPackage%>;

<%for each bImportPkg do%>
//Imported packages
import <%path%>;
<%od%>

//class declaration
<%accessModifier%> <%fieldModifier%> <%name%>
  <%if .bAST.inheritanceExtends != "" %>
    extends <%inheritanceExtends%>
  <%fi%>
  <%for each bIInheritance do%>
    implements <%inheritanceImplements%>
  <%od%>
{

  //Attributes
  <%for each bAttribute do%>
    <%accessModifier%> <%fieldModifier%> <%type%> <%name%>;
  <%od%>

  //Operations
  <%for each bOperation do%>
    <%accessModifier%> <%fieldModifier%> <%returnStmt%> <%name%>(
      <%for each bArgument do%>
        <%type%> <%name%>
      <%od%>
    ){
      <%body%>
    }
  <%od%>
}
<%od%>
```

Chapter 3

Plan Execution

Introduction

We already gave a high-level overview of our research plan in Section 1.3. In this section we discuss those steps in more detail. The execution of our research plan contains the following steps.

1. Selecting UML diagrams and UML tool (Section 3.1).
2. Making a UML profile for more expressiveness (Section 3.2).
3. Transforming XMI into Interformat (Section 3.3).
4. Resolving the stereotypes (Section 3.4).
5. Merging behaviour diagrams with a class diagram (Section 3.5).
6. Transforming Interformat into BAST (Section 3.6).
7. Generating Java source code (Section 3.7).
8. Verifying generated source code with the Java test case system (Section 3.8).

Note that the sequence of the steps differ slightly with the previous presented plan. Instead of discussing the Java test case system (second step in the research plan) as the second step in this chapter, we moved that section to the last step. The reason is because the discussion of the Java system is not needed for the generation process untill we validate the generated source code.

3.1 Selecting UML diagrams and UML tool

We have already mentioned the diagrams that we have selected, which we will use in this project (class diagram, activity diagram, and sequence diagram). We discuss why we have selected this subset in Section 3.1.1. Note that this section gives answer to the research subquestion *'Which behaviour diagrams are suited for generating behavioural source code?'*

The discussion about which UML tool we select can be found in Section 3.1.2, because a UML tool has influence on the UML and XMI version.

3.1.1 Diagram selection

For the selection of diagrams we have studied [DLS+02, DSTW04, KNNZ00, RFW+04] to find out which diagrams are really needed for code generation. The conclusion is that the common diagrams that are used for code generation are class diagram and state diagram, because these two diagrams cover the static and the dynamic part of a system.

However, we miss a diagram in which we can specify the (control) flow of activities. So, we select another diagram that compensates this. We can specify the latter with the sequence diagram and the activity diagram, because we can show activities between various objects with both diagrams. Note that state diagram shows the control flow of states.

We investigated how sequence diagrams are saved. The sequence diagrams are, like activity and state diagrams, saved in two parts. However, unlike those two other diagrams, the sequences do not have an initial node. The sequence of the messages (communication between objects) depends on the layout of a diagram. As a consequence, we have to use the layout of the diagram in order to generate source code. So, we have chosen to use the actvitiy diagram instead of the sequence diagram.

Class diagram As mentioned, a class diagram shows a collection of classes, interfaces, and collaborations and their relationships. Each class contains a class name, class attributes and methods (also mentioned as operations in [BRJ05]). This diagram contains enough information to generate code. We treat the class diagram as a basic diagram.

This diagram has a one to one mapping with the source code. As a result, generation from this diagram is not very difficult. By using this diagram for generation we also satisfy the requirement *'the architecture of the generated code should be the same as the code that is developed manually'* (see Section 1.3).

State diagram Many systems have at least two states (`idle` and `active`). So, engineers can always use a state diagram to model these states. A state diagram can be modelled on all precision levels (architecture, class, method, and attribute). A state diagram is a useful diagram in order to find activities of classes, but it does not contain enough information to generate source code.

For example, a state diagram of an ATM contains a substate `Validating` (see Figure 3.1). When engineers see this state diagram, they know what validation means and what the system has to do. However, a generator has not the same intelligence as humans have. So, the generator is not able to generate code for the substate `Validating`. The generator will be able to generate source code in this situation, if the word `Validating` is a keyword which the generator uses. In addition, the generator must have knowledge of a specific domain (in this case about ATMs), which makes the generator less reusable for other purposes.

Although the state diagram of the example is modelled on the architecture level, the problem of missing information also applies for other levels. We use the do-activity of a state diagram to specify the action within a state.

Activity diagram We want to use the activity diagram to model the body of a method at the attribute level, because this diagram gives us the possibility to model the activity between objects.

The problem with activity diagrams is that abstraction is lost when it is modelled at the attribute level, because the activities contain statements of a particular programming language like `Student student = new Student();`. It seems that this is not the intension of UML, but in [BRJ05] they have shown that it is permitted. However, modelling activity diagram at this (attribute) level 'lies on the edge of making the UML a visual programming language [BRJ05].'

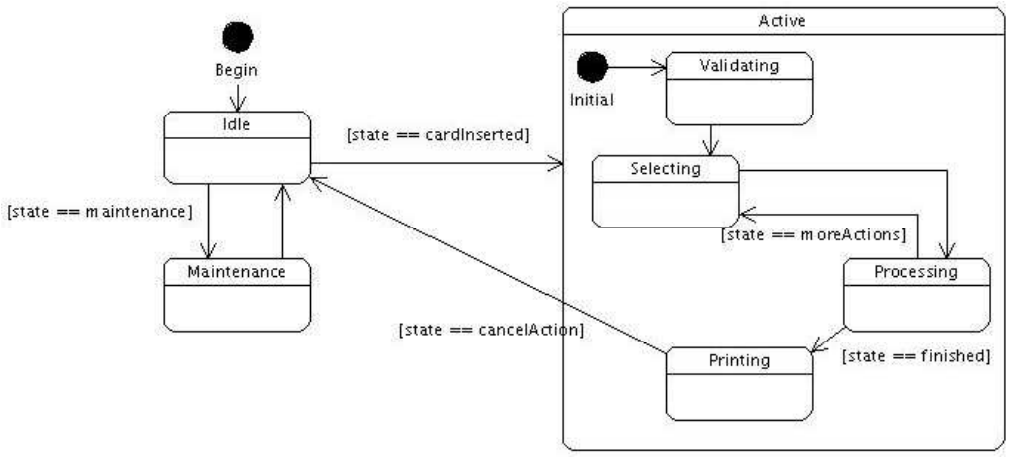


Figure 3.1: State diagram of an ATM, architecture level

3.1.2 UML tool selection

Since the invention of MDA, tools for model transformation and code generation have been developed by many people and companies. A list of MDA tools can be found on [Mod06]. Most of the MDA tools use UML as their modelling language. The difference between an MDA tool and a UML tool is that the first tool supports model transformations (from PIM to PSM), while the second tool does not.

Although we put our project in the context of MDA, we did not choose for an MDA tool. Because this project focuses on the generation from PSM to source code, we will not use the model transformation feature. So, an MDA tool has little added value for this project, therefore we use a UML tool instead.

Requirements while selecting a UML tool

There are more UML tools than MDA tools, because UML exists longer than MDA. A list of all UML tools can be found on [Ano05]. In order to select a UML tool that supports our modelling activities, we made a list of requirements.

1. *The UML tool must execute on a Linux operating system.*
   Linux is our standard operating system on the computers that we use.

2. *The UML tool must have a clear user interface.*
   We do not want to use a UML tool which user interface is difficult to use. For example, if we model a state diagram, we do not want to loose many time to figure out how we can make a orthogonal state and substate.

3. *The UML must not crash every time an error appear.*
   We do not want to use a UML tool that cannot recover from errors. If an error appears and the tool shuts down abruptly, we cannot save our diagrams. So, we want a reliable tool.

4. *The UML tool has to support the selected UML diagrams.*
   If the tool does not support the diagrams that we have selected, the UML tool is not useful to us.

5. *The UML tool has to save the diagrams in XMI.*
   The XMI is, as mentioned, the standard format for saving UML diagrams. So, it is obvious that we want a UML that is able to generate or save their diagrams in XMI. Note that not all UML tools save their diagrams in the XMI format.

6. *A UML tool that supports UML 2.0 is preferred.*
   UML 2.0 is particularly extended to support the modelling and transformations of MDA. It is obvious that we use UML 2.0 as our modelling language. However, many UML tools do not support this version of UML yet. So, we prefer UML tools that support UML 2.0, but it is not a obligatory requirement.

7. *The UML tool must be free of charge, but it does not need to be open source.*
   We want a UML tool that is free of charge, because we do not have a budget in order to buy one.

8. *The UML tool may limit the features, but it must not limit the modelling.*
   A consequence of using free tools of companies, i.e. not open source tools, is that the number of features are limited. We do not mind that a UML tool does not support a copy and paste feature in a free edition, because it does not limit us in modelling. An example of limitation of modelling is that some tools limit the number of elements in a diagram.

Comparing the tools

We have selected a few tools that are open source or free of charge from the list of UML tools, which can be found on [Ano05]. We have compared UML tools whether they satisfy the defined requirements. An overview of the analysed UML tools is in Table 3.1.

Commercial companies have a community edition that is freely available. However, many of them limit the possibility to model diagrams. We found out that VP-UML Community Edition, EclipseUML Free Edition, and Objecteering UML limit its modelling features, but they did not mention it on the website of the company. If we wanted to use a feature that was not a part of the community edition, a pop-up appeared with the question whether we wanted to buy another edition.

We have also installed the open source tools like ArgoUML and Umbrello and evaluated their reliability. However, every time an error appeared, the tools shut down abruptly or did not respond anymore when an error occurred. The project files became corrupted. So, all effort that we put in that project was gone and we had to start all over again.

While comparing the tools we noticed that the free commercial tools have a better reliability than those that are open source. On the other hand, the open source tools have no pop-ups.

Tool that we have chosen

After comparing the tools we have chosen for Gentleware's Poseidon for UML Community Edition. Poseidon is based on the open source and free tool called ArgoUML. The basic version of Gentleware's tool suite, Poseidon for UML Community Edition, is free of charge for non-commercial use. The community edition of this UML tool meets requirement seven, because of the latter.

We have installed Poseidon on our machine and worked with it. Although errors appeared while we were testing the tool, we did not get any annoying pop-ups. Note that the console showed some errors. The most important issue is that our work is preserved without becoming corrupt. So, the tool meets the requirements number one and two.

Poseidon for UML is more feature rich and more stable than ArgoUML. So, Poseidon meets the requirement number three. It is intended for your daily work in commercial and professional environments. ArgoUML, on the other hand, is open source and lends itself for research, study of the architecture and for extensibility. However, we are not using the UML tool as a research object, but as a means that we use to generate XMI.

We can gain an XMI file of our working project in two ways. We can get the XMI file from the project files, which Poseidon saves as a zip file, or we let Poseidon generate the XMI file. Both ways will result in an XMI file with the same content and XMI version (1.2). This feature of saving and generating the content in XMI format meets the fifth requirement.

UML 2.0 is integrated in Poseidon since version 4.0 and it supports the modelling of nine diagrams (see Table 3.1). As a result it also meets the requirements four and six.

The difference between these two is that Poseidon has a better performance, a better graphical user interface, and a different way of saving its diagrams in XMI format. The XMI version in which Poseidon saves its diagrams is 1.2, while diagrams are modelled in version 2.0.

Note that MagicDraw has the same results as Poseidon, but we are more acquainted with the latter. That is why we have chosen for Poseidon instead of MagicDraw.

3.2 Making a UML profile for more expressiveness

We have identified the problems of generating behavioural source code in Section 1.1.

We think that these problems are caused by UML that misses expressiveness. In this section we try to solve these problems by making UML more expressive. In Section 2.2 we discussed the two different ways of extending UML, namely metamodelling and UML profile.

First, we discuss why we have chosen for UML profile to extend UML. Second, we discuss the UML profile that we use during the modelling. Last, we discuss the rules and restrictions that we have beside the UML profile.

Note that this section gives answers to two research subquestions, namely *'How can we constrain the modelling possibilities and simultaneously increase the expressiveness in behaviour diagrams?'* and *'How can we make the mapping between structure and behaviour diagrams clear?'*.

3.2.1 Extending UML: metamodelling or UML profile

[SW06] presents an industrial case study on the choice between metamodel extensions and profiles. They also investigated the influence of the choice on quality of products based on the extensions.

In the paper, they came to the conclusion that there is no single decisive factor whether to choose profiles or metamodels. The most important criteria are the expressiveness of the approaches and the knowledge required for development.

Another conclusion is that customising UML by means of metamodelling is harder than defining a profile. They also present that products based on profiles cause less faults and are of better quality. It also costs less time to define a profile and profiles are easier to develop by many engineers. The benefit of using a metamodel is that the modelling language is more powerfull. It is then possible to define new relations and elements.

Based on the results of this paper we have decided to customise UML with profiles.

3.2.2 Defining a UML profile

In this section we discuss the UML profile that we use. As mentioned in Section 2.2 a profile is a set of predefined stereotypes, tagged values, constraints, and base classes (official definition given in [BRJ05]).

If we use the official definition of a UML profile, then we have only defined several stereotypes. We have not specified any tagged values and constraints, because we do not need them in this project to let our generator generate behavioural source code. We are aware that specification of these parts can be needed in other situations.

UML tool (version)	UML version	Requirements								XMI version	*Comment
		1	2	3	4	5	6	7	8		
ArgoUML (0.18)	1.3	+	+	-	+	+	-	f	+	1.2	We can use all its features for modelling. However, it crashes everytime an error occured
Gentleware Poseidon Community Edition (4.1)	2.0	+	+	+	+	+	+	f	+	1.2	Selected tool. Some features are not supported in this edition, but does not limit the modelling.
BOUML (2.12.1)	1.4	x	-	u	-	*	+	f	+	1.0 1.2	We worked with the tool for a while, but we were not able to model any diagram.
Umbrello Modeller (1.5.1)	1.3	+	+	-	+	+	-	f	+	1.2	
UMLet (6)	u	+	+/-	u	+	+	-	f	+	u	Plug-in for Eclipse
VP-UML Community Edition (5.2)	≤2.0	+	+	+	+	+	+	f	-	1.0 1.2 2.1	Annoying pop-ups about features in other editions and limitation in the modelling
No Magic MagicDraw Community Edition (11.0)	2.0	+	+	+	+	+	+	f	+	1.0 1.1 1.2 2.1	Performance is low.
EclipseUML Free Edition (2.1.0)	2.0	+	+	+	+	-	+	f	-	n	Plug-in for Eclipse.
Violet (0.17.3)	u	+	+	u	-	-	-	f	+	n	Plug-in for Eclipse.
Objecteering UML Personal Edition (5.3.0 SP2)	1.3 1.4	+	+	+	+	+	-	f	-	1.0 1.1	
ESS-Model (2.2)	u	+	-	u	-	+	-	f	+	u	
+ = satisfies the requirement, - = not supported/does not satisy the requirement, f = free edition, u = unknown											

Table 3.1: UML tools which we have analysed for this research.

Although we do not specify UML profile constraints, we do have other constraints. The difference with UML profile constraints and the latter is that we do not use free-form text, and it is only applicable on a higher level. While the UML profile constraints are about specifing the elements and the relations between, our constraints is about limiting the usage of UML elements. In order to avoid the ambiguity of the term *constraint* we use *restriction* instead to indicate our constraints.

First, we discuss the stereotypes that we have defined for our UML profile. Then, we discuss the restrictions that we have defined.

Defined stereotypes and usage

The notation for stereotypes that we use is different from the official one. Instead of defining it as a type which is shown above the name in the format `<< type >>`, we put it as a prefix in a behavioural UML element (activity). We have shown the difference of notation in Figure 3.2 and Figure 3.3.

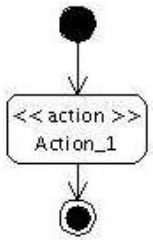


Figure 3.2: Activity with official stereotype notation

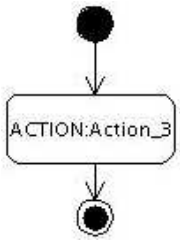


Figure 3.3: Activity with our stereotype notation

Poseidon saves each stereotype that we create as an XMI element. For every other type of UML diagram a whole new set of stereotypes will be created. As a consequence, the XMI contains duplicates of the created stereotypes. An example of duplicated stereotypes is shown in Listing 3.1. Note that ... indicate some omitted data. In order to limit the references to other XMI elements and duplicated stereotypes, we put the stereotypes as the prefix of an action.

Listing 3.1: Example of duplicated stereotypes in XMI

```
...
<UML:Stereotype xmi.id = 'dc6mm7165' name = 'initializer' ... />
<UML:Stereotype xmi.id = 'dc6mm7687' name = 'initializer' ... />
...
```

We have defined five different stereotypes to indicate an action (`ACTION:` and `PRINT:`), a begin (`BEGIN:`), or an end (`END:`). The stereotypes can be used in the activity diagram as well as in the state diagram.

It seems a bit strange to have a `ACTION:` stereotype that can be used for an activity. However, this stereotype has another purpose than an activity on its own. This stereotype is used to collect activities that should execute sequential, like generating new instances for attributes. In this way we can model an action in a consise way. This stereotype has a specific syntax, which is shown in Listing 3.2. Note that we use SDF to specify the syntax.

Listing 3.2: Syntax definition, `ACTION:`

```
"ACTION:" ClassName "." Elements "." ActionActivity -> Action
[A-Za-z\_\$] [A-Za-z0-9\_\$]*                        -> ClassName
"objects" | "attributes" | "methods"                 -> Elements
"GET_METHODS" | "SET_METHODS" | "NEW_INSTANCES"      -> ActionActivity
```

There are some situations where we want to print a variable on the screen, console, or save the information in a file. For these situations we have defined a stereotype `PRINT:` that can be used to print attributes or a

string. In this way, the engineer does not have to give all the statements of the implementation in order to print the specific attribute or string in the activities.

In this project, we only support printing attributes in the console (`System.out.println(""+attributeName.toString());`). The syntax definition for the `PRINT:` tag is shown in Listing 3.3. Note that we omitted some characters and replaced it by ....

Listing 3.3: Syntax definition, `PRINT:`

```
"PRINT:" AttributeName "." PrintActivity  -> Print
"PRINT:" "\"" UDString "\"" "." PrintActivity -> Print
[A-Za-z\_\$] [A-Za-z0-9\_\$]*             -> AttributeName
[A-Za-z0-9 ... \{\}]                      -> UDString
"CONSOLE"                                 -> PrintActivity
```

The `BEGIN:` and `END:` stereotypes are used to specify the modelled level (architecture, class, method, or attribute) of a diagram. These steretypes are used in the activity diagram as well as in the state diagram. Generating code from a diagram that is modelled on the class level is different than a diagram that is modelled on the attribute level. That is why this stereotype is added. The syntax definition is shown in Listing 3.4.

Listing 3.4: Syntax definition, `BEGIN:` and `END:`

```
"BEGIN:" Level                                          -> Begin
"END:" Level                                            -> End
"attribute" | "method" | "class" | "architecture"        -> Level
```

We are aware that there are more possibilities with stereotypes than we use and have in our UML profile. We could, for example, make a stereotype `EXCEPTION:` for exception handling or `CAST:` to cast a particular object to another. However, for this project we only need these types. In order to make this UML profile useful for other research in the future, we take the extensibility of the UML profile into account, while we defined the syntax.

In a later step we will use this syntax definition in order to transform these tags in Java source code (see Section 3.4).

Diagrams that are modelled with our UML profile are mentioned as annotated diagrams in our project. An example of an annotated diagram is shown in Figure 3.4.

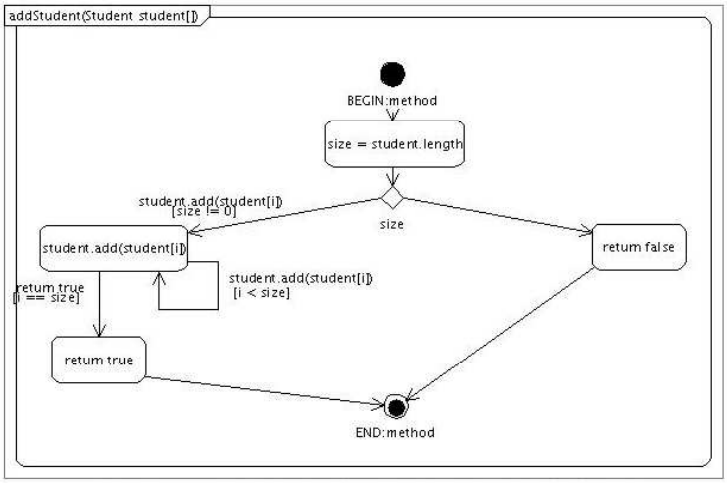


Figure 3.4: Activity diagram modelled at the attribute level

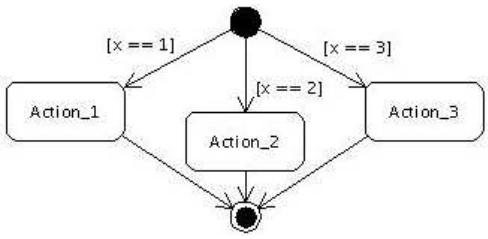


Figure 3.5: An activity diagram that is ambiguous

Restrictions in the modelling

As mentioned before, the difference between the UML profile constraints and these restrictions is the level on which they are applicable. The UML profile constraints is about specifing the elements and the relations between them by using operators (e.g. `xor`, `or`, `and`) and conditions (e.g. `person1 == "male" && person2 == "female"`), while our restrictions are about limiting the usage of UML elements by not allowing every construction in a diagram.

There are many ways possible to model a particular condition or sequence. Because of that, diagrams can be ambiguous. An example of a diagram that is ambiguous is shown in Figure 3.5. The figure shows three different activities (`Action_1`, `Action_2`, and `Action_3`) and three relations which trigger them. There are three implementation possible with this diagram. (1) Only one condition can be true at a time, so only one activity will execute. (2) `x` will increase or decrease each time after executing an activity, so these activities are executed sequential. (3) `Action_1`, `Action_2`, and `Action_3` are executed at the same moment (parallel execution).

A generator cannot generate source code (or it generates bad source code) from an ambiguous diagram. By restricting the usage of the UML elements, we eliminate the problem *'a behaviour can be implemented in different ways'* that we have defined in Section 1.1. A benefit of this kind of restrictions this is that we always know when a conditional sequence stops. The restrictions that we have are defined as follows.

- *The diagram must be saved at the same level and have the same name as the related UML element.*
  It is not possible to relate a diagram to a method, a class, or an attribute in Poseidon. So, by giving the behaviour diagram the name of the related UML element and save its position at the same level, we can map the diagrams with the structure diagram.

  An example of this restriction is shown in Figure 3.6. The Figure shows a project tree that is opened in Poseidon, which contains four packages, one class, and two activity diagrams. Both activity diagrams are saved as a leaf of the class `School`. The unfolded activity diagram has the same name as the method `showStudents()` in the same class.

  If we do not use the position of the diagrams for the mapping between structure and behaviour diagram, we need to give the diagrams the qualified name of the UML element. In the case of Figure 3.6, the activity diagram must be named as `school.School.showStudents()`. Note that the diagram's name must also include the arguments of the method. The reason is because of the possibility of overloading in Java. In this way we prevent generation of behaviour source code for the wrong method.

  This restriction solves the mapping problem *'the mapping between structure and behaviour diagrams is not always clear'*.

- *Each state must contain an action in the do-activity section.*
  We can generate source code from a state diagram that contains states without actions, but the resulting source code will not contain behaviour. That is why behaviour or the activity of the state must be indicated with the do-activity (is shown as `do/` in the diagram).

- *Every activity and state diagram has only one initial node/state and has one or more final node/state. Except for composite state, which has only one initial and one final state.*
  By limiting the initial node/state in each diagram, the generator knows where to start with generation of source code. For example, if an activity diagram is modelled on the attribute level, which is a execution model of a method, it cannot have two initial nodes, because a method execution will always start at the beginning.

  A final node/state is needed to know when an activity or state sequence is ended.

- *An activity may have only one outgoing relation, but it may have a conditional relation to itself.*
  In order to prevent ambiguity like shown in Figure 3.5, an activity may only have one outgoing relation. Except for a conditional relation to itself, because this relation indicates a repetition of the same action.

- *A decision element contains one or more conditional relations and may contain zero or one relation without a condition.*
  A decision indicates a conditional execution. It is possible that a decision element has multiple conditional relations, because now we know that it is really a conditional execution instead of a parallel execution. Because of that, it is also possible to have relation without a condition, which will be treated as the default execution if nothing else is true.

- *A conditional relation (a guard) can only be used after a decision element. It is not allowed to use a conditional relation between two activities.*
  This restriction make decisions visible in diagrams. Although it seems like this rule overdo it when there is only one conditional relation between activities, it is needed for consistency with conditional execution.

- *The sequence of a decision element (a shape of a diamond) must always be terminated with a final element (a circle within a circle) or a merge element (also a shape of a diamond).*
  This restriction takes care of the fact that the conditional execution should be terminated in a well-formed way. An example of a good termination is shown in Figure 3.7.

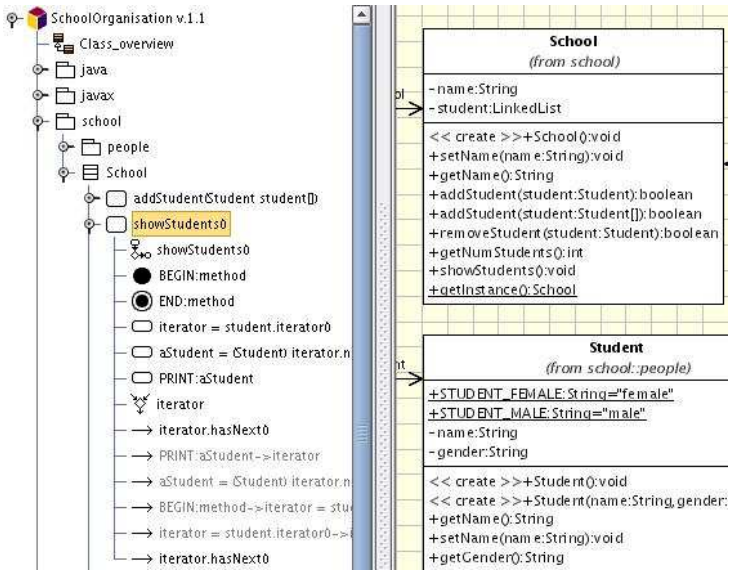


Figure 3.6: Position of an activity, as leaf of a class node in the project tree

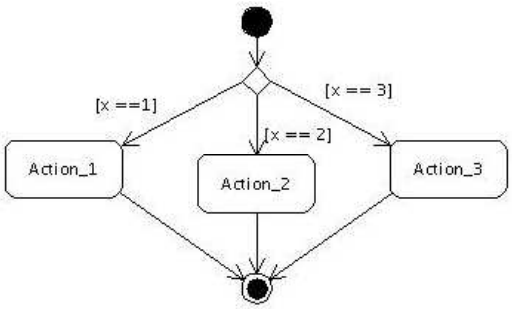


Figure 3.7: An example of a conditional execution that is terminated in a well-formed way

Results and consequences of using a UML profile

Activity diagrams and state diagrams, which are modelled by means of the UML profile on the attribute level, have a one to one mapping with the source code. Althought a state diagram still needs an activity diagram that specifies the states, we are able to map the states to source code. We will discuss the latter in Section 3.5.2.

The restrictions eliminate ambiguous interpretation of the diagrams and make the mapping between structure and behaviour diagrams clear. As a result of having these restrictions, the implementation of the transformer, which transforms diagrams into source code, is less complex, because the generator does not have to take every possibility into account.

A consequence of using a UML profile is that we have to make a transformer that transforms the stereotypes to source code. For example, the stereotype `PRINT:` contains the following data `"Hello World".CONSOLE` will

generate `System.out.println(""+ "Hello World");`. The discussion about this transformer can be found in see Section 3.4.

3.3 Transforming XMI into Interformat

In this step we transform the XMI of Poseidon into our own reduced XMI format ("Interformat"). Transforming the XMI to Interformat is the first step of our generator that we build for the generation of behavioural source code.

We will first explain why we have decided to make this transformation. By explaining this transformation we also give answer to the research subquestion *'How are behaviour diagrams saved in XMI, and is that representation practical for code generation?'* We already answered the first part of this question in Section 2.4 for the activity diagram and state diagram. So, we will try to answer the second part of this research subquestion. After this we discuss our implementation of the transformer.

3.3.1 Why we make this transformation

As mentioned before, there are many tool-specific versions of XMI. Transformers and generators, which use these XMI for their own purpose, are also tool-specific. Although it depends on the architecture of transformers and generators, it makes them less reusable for other projects that use other UML tools. In this case, we have to make the same tools for every different tool-specific XMI. If the UML tool changes the syntax of its tool-specific XMI, then we have to modify our toolset. It is not hard to imagine that it will take a lot of effort to change all the affected tools.

An XMI file contains many references to other elements inside the document, but it can also have references to elements that appear in other documents. This makes code generation difficult from this format, because it has to solve these references first before it can generate code, while the documents possibly are generated by two different tools.

The XMI that is generated by Poseidon (our selected tool) is very verbose. XMI elements contains attribute names and variables that are not useful for code generation. Because of the latter, an XMI can easily expand to megabytes, which is a problem for all UML tools. The syntax of the XMI, which is defined by OMG, is the cause of this consequence. The latter is discussed in Appendix A.

An example of a XMI element that contains not neccessary information is shown in Listing 3.5. A package in Java is actually nothing more than a file that contains classes. We only need the name of the package in order to create that file. So, the other attributes are useless.

Listing 3.5: XMI package element generated by Poseidon

```
<UML:Package xmi.id = 'I112202m10b37737dc6mm7152' name = 'school'
  visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false'>
```

The XMI files that we used during this project are mostly over four megabytes, while those XMI files only contains a few diagrams. We have tested a case that contained more than sixty classes. We let Poseidon import the classes (Java source code) and created one class diagram. The XMI file of that project was over fifty megabytes. Poseidon could not process the XMI file and it caused a memory error (stack overflow).

Our conclusion is that XMI is not the best format to use for code generators, because (1) using this format makes the generator not reusable for other XMI files, (2) it can contain references to other documents, which are generated by other UML tools, (3) it is very verbose, and (4) it can expand to megabytes that lead to memory errors.

So, we developed an intermediate format, which the tools will have to use for code generation. The intermediate format saves the XMI in a compact way. The format should only contain data which is useful for

the generation process, so that the size of the document will be limited. References to other documents will be processed, before the generation of code.

The tools that we develop are tool-independent, because they use the intermediate format instead of the tool-specific XMI. Although we have to make a transformer for every tool-specific XMI that transforms it into our intermediate format, the benefit is that the changes in the tool-specific XMI has only effect on those transformers.

3.3.2 The intermediate format: Interformat

We decided to keep the intermediate format similar to the XMI that is generated by Poseidon, because there are several free tools available to modify XML-based formats, which we can use for the transformation from XMI to Interformat.

The biggest differences are (1) the number of different elements to save its data, (2) the number of different attributes, (3) the names of the attributes, (4) layout information is omitted in our format, and (5) the elements are saved in a particular sequence.

The format that we have cannot be parsed with the official XMI definition, because elements that do not contain any important information like `<UML:Namespace.ownedElement>` and `<UML:TaggedValue. type>` will be omitted. We keep only one XMI element of this type to indicate the behaviour diagrams, namely `<UML:BehaviorDiagramContent>`. In this way we can easily traverse through the document for behaviour diagrams by matching directly with this element.

We made a syntax definition of the Interformat in order to check the correctness of the generated document. The architecture (module view) of our Interformat is shown in Appendix B next to the architecture of XMI (specified by OMG). Note that the Interformat is actually a simplified version of XMI.

3.3.3 Transforming XMI into Interformat

Manually rewriting the XMI generated by Poseidon to the Interformat by hand is very error prone and labour-intensive. So, we made a transformer for this rewrite phase. We had several possibilities to implement this transformer. We could implement it in ASF+SDF or in XSLT. Althought there were many other techniques which we could use for this implementation, the mentioned two are the most obvious ones.

XSLT or ASF+SDF

This transformer transforms the XMI into another XML-based format. We can implement it in XSLT, which is particularly developed to transform XML-based documents, or in ASF+SDF, which makes geneneric (language parametric) transformation possible.

If we implement this transformer in XSLT instead of AFS+SDF we gain four benefits. (1) We do not have to take the syntax of the target code into account. (2) We do not have to write much code that does the transformation. (3) If we want to change the syntax of the target code, we can easily modify it in the template. (4) It will take us less time to develop this transformer in XSLT than ASF+SDF, because XSLT is especially developed to transform XML-based documents, while ASF+SDF has no special features at all for these documents.

That is why we have chosen to use XSLT as implementation technique for this transformer.

Implementation of the transformer

We divided the transformer in four different modules which are divided over three layers. The first (top) layer contains one module which gives access to the different main aspects of an XMI document. The second layer contains one module with the purpose to transform general XMI elements. The third layer contains two modules with the purpose to transform behaviour diagrams content. The architecture of the transformer is shown in Figure 3.8. By separating the transformer in several different parts, we can easily extend it.

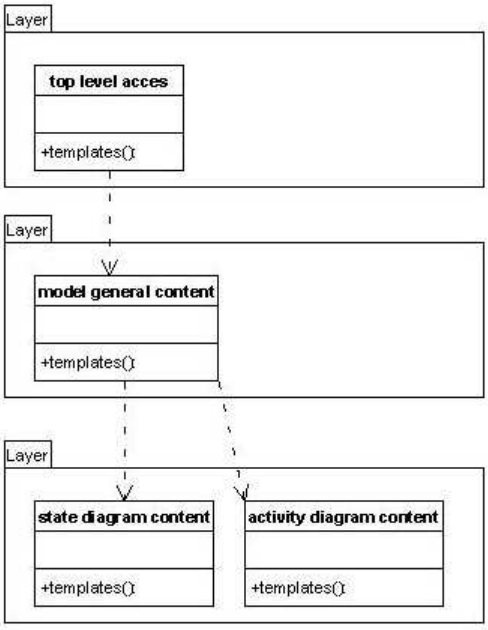


Figure 3.8: Architecture of the first transformer

We only selected the elements and attributes which are important for the final target code. We tried to reuse the transformation code as much as possible. The target source code is validated by focussing on the different forms of those elements at the possible places to check whether it will work in those circumstances. A small part of our transformer is shown in Listing 3.6.

Listing 3.6: A small code snippet of the XMI to Interformat transformer

```
<xsl:template match="UML:Model">
  <xsl:element name="UML:Model">
    ...
    <xsl:apply-templates select="UML:Namespace.ownedElement" />
  </xsl:element>
</xsl:template>

<!-- UML:Namespace.ownedElement element -->
<xsl:template match="UML:Namespace.ownedElement">
    ...
    <xsl:for-each select="UML:Package">
      <xsl:call-template name="UML:Package" />
    </xsl:for-each>
    <xsl:for-each select="UML:Class">
      <xsl:call-template name="UML:Class" />
    </xsl:for-each>
    ...
</xsl:template>
```

```
<!-- UML:Class element -->
<xsl:template name="UML:Class">
  <xsl:element name="UML:Class">
    <xsl:attribute name="xmi.id">
      <xsl:value-of select="@xmi.id" />
    </xsl:attribute>
    <xsl:attribute name="name">
      <xsl:value-of select="@name" />
    </xsl:attribute>
    ...
    <!-- This assignment reuses previously defined tag -->
    <xsl:apply-templates select="UML:Namespace.ownedElement" />
    ...
    <xsl:if test="UML2:BehavioredClassifier.ownedBehavior">
      <xsl:call-template name="UML:BehaviorDiagramContent" />
    </xsl:if>
  </xsl:element>
</xsl:template>
```

3.3.4 Gained results

As mentioned, the Interformat is a simplified XML-based document, which is similar to the XMI specification. XMI elements that has no attributes, like `<UML:Namespace.ownedElement>`, are omitted. Only one element is kept in the Interformat to indicate the behaviour diagrams, `<UML:BehaviorDiagramContent>`.

We merged some tags, like the elements which represents the guard between relations. An example of the latter is shown in Listing 3.7 (XMI) and Listing 3.8 (Interformat). As a result, the Interformat has less references to other XMI elements. It makes transformation of behaviour diagrams easier, because we do not have to traverse much further through the tree for information.

Listing 3.7: XMI element of a guard in an activity diagram

```
...
<UML2:ActivityEdge.guard>
  <UML2:OpaqueExpression
    xmi.id = 'mm76a1'
    name = 'gender == STUDENT_FEMALE'
    visibility = 'public'
    isSpecification = 'false'
    body = 'true'
    language = 'java'/>
</UML2:ActivityEdge.guard>
...
```

Listing 3.8: Interformat element of a guard in an activity diagram

```
...
<UML:ActivityEdge.guard
  xmi.id="mm76a1"
  name="gender == STUDENT_FEMALE"
  visibility="public"
  isSpecification="false"
  body="true" />
...
```

An example of the Interformat is shown in Listing 3.10. The corresponding XMI, which is generated by Poseidon, is shown in Listing 3.9.

We compared the two different formats and the results are presented in Table 3.2. As the table shows the Interformat is only 14.29% of the original XMI size.

Listing 3.9: XMI generated by Poseidon

```
<UML:Package xmi.id = 'I112202m10b37737dc6mm7152' name = 'school'
  visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false'>
  <UML:Namespace.ownedElement>
```

```
<UML:Package xmi.id = 'I112202m10b37737dc6mm7143' name = 'people'
  visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false'>
  <UML:Namespace.ownedElement>
    <UML:Class xmi.id = 'I112202m10b37737dc6mm716f' name = 'Student'
      visibility = 'public' isSpecification = 'false'
      isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
      isActive = 'false'>
      <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id = 'I112202m10b37737dc6mm7142'
          isSpecification = 'false'>
          <UML:TaggedValue.dataValue>false</UML:TaggedValue.dataValue>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref = 'I112202m10b37737dc6mm7151'/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
        <UML:TaggedValue xmi.id = 'I112202m10b37737dc6mm7134'
          isSpecification = 'false'>
        ...
```

Listing 3.10: Interformat

```
<UML:Package xmi.id="I112202m10b37737dc6mm7152" name="school">
  <UML:Package xmi.id="I112202m10b37737dc6mm7143" name="people">
    <UML:Class xmi.id="I112202m10b37737dc6mm716f" name="Student"
    visibility="public" isLeaf="false" isAbstract="false">
      <UML:ModelElement.tagdefinition xmi.idref="I112202m10b37737dc6mm7151" />
      <UML:ModelElement.tagdefinition xmi.idref="I112202m10b37737dc6mm713e" />
      ...
```

	Poseidon's XMI	Interformat
Number of lines	65,386	9,344
Source file size in kb	4,176.26	608.5

Table 3.2: Results and difference between Poseidon's XMI and our Interformat

3.4 Resolving the stereotypes

A consequence of using a UML profile is that we have to make a transformer that transforms the stereotypes and tagged values into source code. We only have to deal with the stereotypes in our case, because we did not specify any tagged values.

This transformer will not change the structure of the interformat document. It replaces the generated Java source code with the original attribute value. For example, the transformer gets a call node with the stereotype `ACTION:Students.objects.NEW_INSTANCES`, it transforms all objects of the class Student into `Student student = new Student();`. Note that there is only one object of the class Student in this case.

This transformation is a step before the merging step. If we process the stereotypes while we are merging, the implementation of the merger will be very difficult, because we have to take two activities (processing stereotypes and merging) into account. We make the next step easier by transforming the stereotypes into Java source code before the merging step.

The path of this transformation is shown in Figure 3.9.

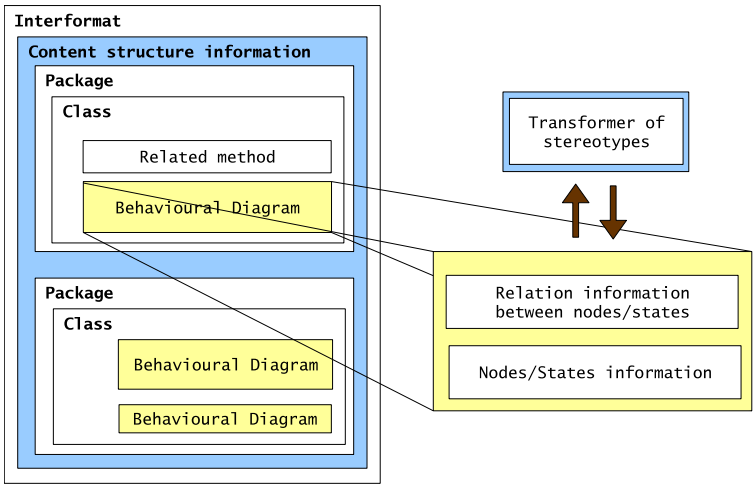


Figure 3.9: Transformation path of this step

XSLT or ASF+SDF

We have chosen to implement this transformer in ASF+SDF, because it is easier to modify strings (attribute values in this case) with ASF+SDF than with XSLT. Although ASF+SDF does not provide a standard set of functions, which we can use to modify the strings, we can use the syntax definition of the stereotypes that we have defined in Section 3.2.2. Note that we did not implement this transformer, because we had too little time. However, we performed this transformation step manually.

3.4.1 Transforming the stereotypes into Java code

Because every annotation has a fixed notation and options, we can easily make a transformer for this UML profile. However, the difficulty of implementing this transformer is the validation part.

The transformer takes a string as input, which is in this case a value of an attribute (to be more exactly an XMI-value `<UML:ActivityNode xmi.value="an XMI-value">`). Depending on the given annotation assignment, it transforms some Java source code. When an XMI-value is not an annotation, then it will return the same XMI-value back.

We have to validate for every `ACTION:`-tag the reference to a class whether it exist or not. Depending on the next argument (`objects`, `attributes` or `methods`), we gather all defined objects, attributes and methods. Then we have to process a particular action type (`GET_METHODS`, `SET_METHODS`, `NEW_INSTANCES`) on every attribute or method.

If an activity node contains `Student.attributes.GET_METHODS` it will make a `getXXX()` method for every attributes in the class `Student`. This annotation can only processed if its diagram is a part of the class `Student`.

The `PRINT:`-tag prints for every given string on a screen (like in a new window) or a system (like in a console). Depending on which option it is written, the transformer generates source code that prints the string on the screen by making a new window or by just returning `System.out.prinln(<string to be printed>);`.

3.5 Merging behaviour diagrams with a class diagram

We discuss the merging activities of behaviour diagrams with a structure diagram in this section. This merging process is the second step of our generator in order to generate behavioural source code. Our

generator use an Interformat document as its input. Note that the Interformat document still contains structural (class) diagram and the modelled behaviour diagrams.

Code generation from a behaviour diagram is difficult, because there are many interpretation possible. However, generating behavioural source code from a structure and one or more behaviour diagrams at the same time is very difficult. A consequence is that the implementation of the generator would be very complex.

Generation of source code with behaviour diagrams can be made easier by merging them into the class diagram. We transform each behaviour diagram into Java source code. After each transformation we place the source code in the right structure as a part of the class diagram. In the end, the Interformat document contains only the class diagram.

This merging of the diagrams is possible because of the restriction that we have defined (see Section 3.2.2): *'The diagram must be saved at the same level as the related UML element and its name'.*

In this section we show that we can map the behaviour diagrams, those which are modelled on attribute level, to source code by means of our UML profile. The mentioned activities of this step are shown in Figure 3.10.

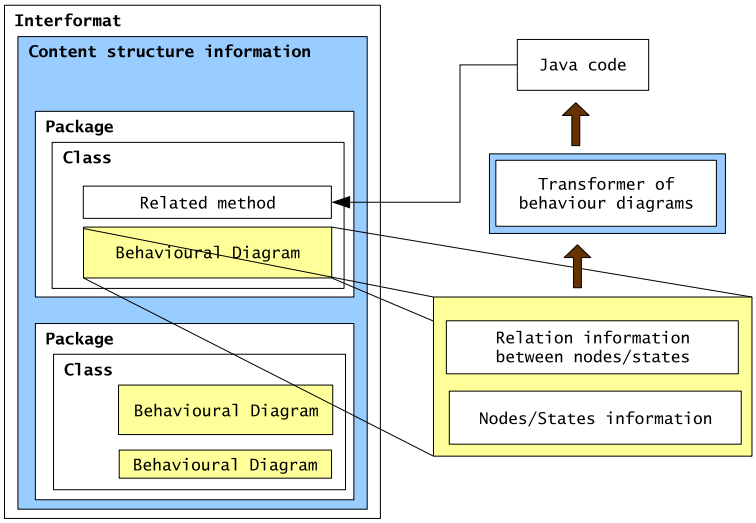


Figure 3.10: The activity of the merge - merging the behaviour diagrams into the class diagram

3.5.1 Transforming an activity diagram

When all stereotypes of a diagram are transformed, we are going to transform the behaviour diagrams into Java source code. We discuss the transformation of an activity diagram first. We will show a one to one mapping between an activity diagram and the source code in this section.

Note that we did implemented this transformer, which is also succeeded to generate behavioural source code. However, the implementation of the generator is not finished yet. At this moment we can generate behavioural source code from behavioural diagrams, but the feature that replaces the value of the attribute `body` in the interformat with the generated behavioural source code still has to be implemented. The latter is performed manually.

How activity diagrams are structured

Like the activity diagram in the original XMI, this activity diagram is also separated in two parts. Those parts have almost the same name, namely `ActivityEdge` and `ActivityNodes`. The `ActivityEdge` contains

activity edges (relations between activities), while the `ActivityNodes` contains activity nodes (activities).

Each activity edge contains information about the activity that triggered the relation (`source`), the target activity (`target`), and a possible guard (`guard`). An example is shown in Listing 3.11

Listing 3.11: An example of an activity edge

```
<UML:ActivityEdge xmi.id="mm76cf" name="" ...>
  <UML:ActivityEdge.source xmi.idref="mm76e9" />
  <UML:ActivityEdge.target xmi.idref="mm76f7" />
  <UML:ActivityEdge.guard xmi.id="mm76a1" name="gender==STUDENT_FEMALE"
    visibility="public" isSpecification="false" body="true" />
</UML:ActivityEdge>
```

There are six different types of activity nodes. Each type contains information about the incoming and outgoing edges, and is a representation of a building block of UML. There are three different types of building blocks shown in Figure 3.13. Note that there is only one type of building block (the rounded rectangle) that contains a context related activity.

Generation of source code

An engineer can express all control constructs (`if-then`, `if-then-else`, `while-loop`, `do-while`, `switch-case`, and `for-loop`) in the activity diagram. However, the generator will not generate exactly the same constructs for `do-while`, `switch-case`, and `for-loop`. We give you a few examples of the mapping between the modelled constructs and the generated source code.

A `switch-case` can be modelled like shown in Figure 3.11. The generator generates an `if-then-else` instead of a `switch-case` (see Listing 3.12). As a consequence the generated source code contains duplicate code, but the modelled behaviour is preserved. Note that the decision block is closed with a final node, but it can also be closed with a merging node.

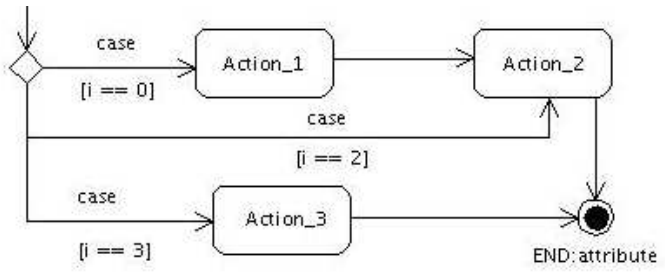


Figure 3.11: A `if-then-else` expressed in an activity diagram

Listing 3.12: Mapping with `if-then-else`

```
if( i == 0 ){
  //Action_1
  //Action_2
}else if( i == 2 ){
  //Action_2
}else if( i == 3 ){
  //Action_3
}
```

We modelled a `do-while`, which is shown in Figure 3.12. The generator generates source code that is shown in Listing 3.13. The reason why this model gives a `while-loop` is because the activity `Action_1` has a relation to itself. The guard of the relation is used as the expression of the `while-loop`. To keep the same behaviour as a `do-while`, we also put the *//Action_1* before the `while-loop`.

A `while-loop` is shown in Figure 3.13, which will result in source code shown in Listing 3.14. The difference between this figure and the previous one is that the relation after the decision node has a condition (guard). Every relation after the decision node that has a condition will result in an `if-then` statement. That is also the reason why the `while-loop` is nested in an `if-then`.

The result of this transformation will be saved as an attribute value of an Interformat element (`<UML:Operation.body ... body="Java source code"/>`). Note that we validated this transformation with several activity diagrams in order to check that the generator is implemented well.

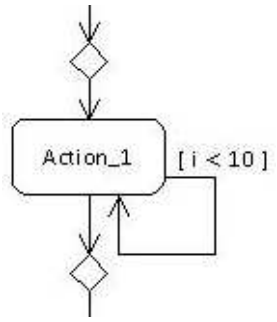


Figure 3.12: A `do-while` expressed in an activity diagram

Listing 3.13: Mapping with `do-while`

```
//Action_1
while( i == 0 ){
  //Action_1
}
```

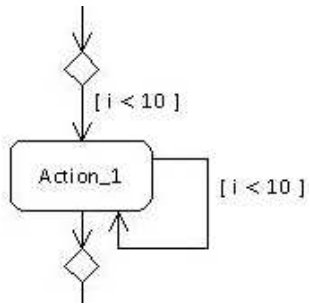


Figure 3.13: A `while` loop expressed in an activity diagram

Listing 3.14: Mapping with `while`

```
if( i < 10 ){
  //Action_1
  while( i < 10 ){
    //Action_1
  }
}
```

3.5.2 Transforming a state diagram

A state diagram cannot be used on its own to generate code, because otherwise the generator should contain keywords with its implementation (source code) as mentioned in Section 3.1.1. For the sake of reusability of the generator in other projects or other state diagrams with other states, we did not use keywords for states. As a consequence, the state diagrams have to have references to activity diagrams.

Note that we did not implement this transformation. We will write this section as if we did implement this part. The validation of the generated behavioural source code is done by hand.

How state diagrams are structured

Just like the activity diagram, the state diagram also consists of two parts, `Region.subvertex` and `Region.transition`. The `Region.transition` contains transitions (relations between states), while the `Region.subvertex` contains the states.

The transitions have the same type of information as an activity edge, namely a source, a target, and a guard (if needed).

As shown in Section 2.4.2 state diagrams are more complicated than activity diagram. A state can be modelled as a simple state, a substate, and a composite state. Each state can contain information (advanced features) including the effect when entering or exiting a state, and internal transitions.

The transformation of state diagrams will result in Java source code. The Java source code will be placed as an attribute value of its related Interformat element.

Generation of source code

We distinguish the state diagrams in two groups, namely state diagrams that have relations with guards, and state diagrams that have relations without guards. While a state diagram without conditional relations can be mapped to a `switch-case`, state diagrams with conditional relations can be mapped into an `if-then-else` statement.

An example of a state diagram without conditional relations is shown in Figure 3.14. Note that the states contain do-activities. These activities are executed when the state is active. We used the stereotypes that we have defined in our UML profile. Our generator will generate a `switch-case` that is shown in Listing 3.15.

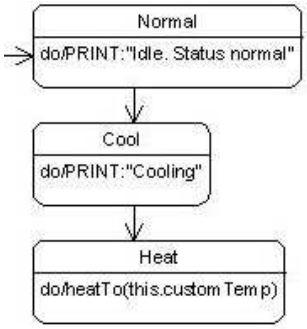


Figure 3.14: A `switch-case` expressed in a state diagram

Listing 3.15: Mapping with source code (`switch-case`)

```
switch(status){
case NORMAL:
  System.out.println("Idle. Status normal");
  break;
case COOL:
  System.out.println("Cooling");
  break;
case HEAT:
  heatTo(this.customTemp);
  break;
}
```

An example of a state diagram with conditional relations is shown in Figure 3.15. The generator generates an `if-then-else` statement with the defined states within. An example of the latter is shown in Listing 3.16.

An interesting question is what the generator will generate if we combine these two diagrams together. The answer is that the generator will generate an `if-then-else` statement with the defined do-activities within. As a result we will get source code as shown in Listing 3.17.

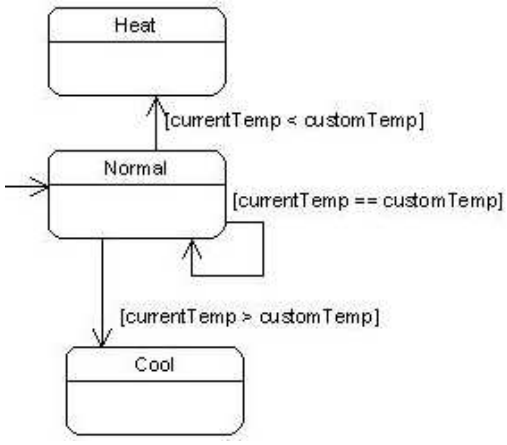


Figure 3.15: An `if-then-else` statement expressed in a state diagram with the defined states

Listing 3.16: Mapping with `if-then-else`

```
private final static int NORMAL = 0;
...
private int status = NORMAL;

...

  if(currentTemp < customTemp){
  status = COOL;
  }else if(currentTemp > customTemp){
    status = HEAT;
  }else{
    status = NORMAL;
  }
```

Listing 3.17: Mapping with source code of a state diagram that has relations with and without conditions

```
if(currentTemp < customTemp){
  status = COOL;
```

```
  System.out.println("Cooling");
}else if(currentTemp > customTemp){
  status = HEAT;
  heatTo(this.customTemp);
}else{
  status = NORMAL;
  System.out.println("Idle. Status normal");
}
```

3.5.3 Implementation of the merger

We discuss the drawbacks and the benefits of the two technologies (ASF+SDF and XSLT) here, which is specific for this transformation step.

XSLT or ASF+SDF

We have chosen to implement the merger in ASF+SDF, because it take us less effort than in the case of XSTL. If we would implement it in XSLT we have to write many duplicate code, while it is not needed in ASF+SDF.

Both behaviour diagrams are divided in two parts, namely a part with relations and a part with the states or nodes. These two parts contain references to each other. The generator has to remember the values of the visited attributes each time it passes a state or a node and merge them together.

Initially, we wanted to implement the merger in XSLT. However, the variable values of XSLT can only remember one value while processing. As a consequence, we have to make new variables in order to save the attribute values each time we pass a relation or state/node, which is not very efficient.

It is not possible to give arguments with the defined templates in XSLT. As a consequence, we have to duplicate the code in order to process the diagram.

XSLT gives a limited set of functions to modify the characters of a particular value. However, we need a more sophisticated set of functions for this transformation. It is possible to extend the functions in XSLT, but then we probably have to change our XSLT parser.

If we implement it in ASF+SDF, we do not need another processor and a large set of different variables to remember the values separately. So, the implementation contains minimal duplicate code.

In ASF+SDF there are no predefined functions to modify characters of a lexical sort, except for the lexical constructor functions. However, it is enough to make a set of functions that is specific for this transformation. The benefit is that we will have a set of functions that is very useful, however the drawback is that it will take us time to perform these transformations.

3.6 Transforming Interformat into BAST

The template-based generator of [Arn06] uses one or more templates to generate source code. The templates must be created by ourselves and conform to the syntax of the target programming language, in this case Java. The generator parse the given template(s) before the generation process and checks whether the templates are created in a well-formed way. As a result of the latter, we can ensure that the generated code can be compiled by the Java compiler.

Like every template, it contains holes that must be filled with data from a source. The data source of the template-based generator is a ‘Balance Abstract Syntax Tree’ (BAST). The template-based generator gets the data from the tree by means of the expressions that are given in the templates.

The reasons why we use this generator are: (1) the generator already exists, (2) it is easier to transform the Interformat to BAST than to create a generator of our own, (3) this generator can ensure that the syntax of generated source code is correct, and (4) we do not have to put much effort in learning the Java syntax.

In this section we discuss the transformation from Interformat to BAST. Note that this is the last transformer that we would make for this project. This transformer is, however, not implemented, because we do not have enough time to realise it.

3.6.1 Extending the syntax definition of BAST

As mentioned in Section 2.6.1 the BAST does not contain enough sorts to let us express other parts of a class. Fortunately, the template-based generator is build very flexible. We can add new sorts to the syntax definition and let it traverse through the tree without modifying the behaviour of the generator.

We added five new sorts to the BAST syntax. Those are `BInterface` for interfaces, `bImportPkg` for the imported packages in a class, `bIInheritance` for inheritance of interfaces, `bOperation` for methods, and `bArgument` for arguments of the methods. The syntax of these sorts is given in Listing 3.18.

Listing 3.18: Extended syntax for BAST

```
"bInterface" "(" "[" {BClassItem ","}* "]" ")"       -> BInterface

"bImportPkg" "(" "[" { BProperty ","}* "]" ")"      -> BImportPkg
"bIInheritance" "(" "[" { BProperty ","}* "]" ")"   -> BIInheritance
"bOperation" "(" "[" { BArgument ","}* "]" ")"      -> BOperation
"bArgument" "(" "[" { BProperty ","}+ "]" ")"       -> BArgument

BOperation | bImportPkg | bIInheritance   -> BClassItem
```

3.6.2 Implementation of the transformer

Unlike the Interformat, the BAST format (see Section 2.6.1) has only four sorts (string, int, real, bool) in which we can save our content information. In order to let those types make more sense in the BAST, we can give a specific name to the `BProperty`. We will use as much as possible the names that are used in the Java specification to incidate the specific parts. An example of this is a `BProperty` that is constructed as follows: `type(str("Student"))` or `name(str("name"))`.

Because of the limited expressiveness of the BAST format, the resulting document will be very verbose. The benefit is that the document does not contain references to other elements any more. So, the information, which is needed to generate code, is more localised. As a result, the template-based generator does not have to be very smart.

An example of the resulting BAST format is shown in Listing 3.20. The source file, from which the BAST is generated, is shown in Listing 3.19.

Listing 3.19: Snippet of the Interformat

```
...
<UML:Package xmi.id="I112202m10b37737dc6mm7152" name="school">
  <UML:Package xmi.id="I112202m10b37737dc6mm7143" name="people">
    <UML:Class xmi.id="I112202m10b37737dc6mm716f" name="Student"
    visibility="public" isLeaf="false" isAbstract="false">
      <UML:Attribute xmi.id="I112202m10b37737dc6mm6fbf"
      name="STUDENT_FEMALE" visibility="public" ownerScope="classifier"
      changeability="frozen" xmi.idref="I112202m10b37737dc6mm70f6"
      initialValue="&amp;quot;female&amp;quot;" />
...
```

Listing 3.20: Interformat that is transformed into BAST format

```
...
bClass([
  name( str("Student") ),
  containedPackage( str("school.people") ),
  accessModifier( str("public") ),
  fieldModifier( str("") ),
  inheritanceExtends( str("") ),
  bAttribute([
    name( str("STUDENT_FEMALE") ),
    accessModifier( str("public") ),
    fieldModifier( str("static") ),
    type( str("String") ),
    initialValue( str("\"female\"") )
  ])
  ...
])
...
```

XSLT or ASF+SDF

Because the BAST and the template-based generator are developed in ASF+SDF, it is obvious that we implement the transformer in the same technology. By doing so, we also can ensure that the generated BAST format is correct.

3.7 Generating Java source code

We discuss the templates that we make for the generation of Java source code in this step. We will also show the generated source code from the template.

The BAST that this generator uses is generated in the previous step from an Interformat. Note that this generator is made by [Arn06].

3.7.1 Creating the templates

The templates that the generator uses contain holes. The expressions in the holes matches with a given custom name or a sort. The generator knows how to traverse through the BAST by means of the expressions. The generator returns an element or elements by giving the path (the expression) of a particular element.

We do not have to put much effort to learn and use it, because the language to let the generator traverse through the tree is quite similar to XSLT. The difference between XSLT and the expressions that are used in the templates is that the latter will not parsed as a transformation language, but as a part of the target language.

It is possible to use the built-in control constructs of the generator like `<% for-each ... %>`. In this way we can repeat the same action without defining it again. An example of a template is shown in Listing 3.21. Note that this template contains many holes. It makes the template less human readable. However, it makes the template much more reusable, because it contain less source code of the target language.

We can only generate classes with the given example Listing 3.21, whether they are abstract or not. However, Java also has interfaces. So, we have to make a template for interfaces. However, this template is similar to the template that is used to generate classes. The biggest differences are multiple inheritance of other interfaces and methods without body statements.

Listing 3.21: Template for generating a Java class

```
<%for each bClass in bAST do
  //Save each class in a file as follows <class name>.java
  store in <%.bAST.name%>/<%name || ".java"%>
%>
package <%containedPackage%>;

<%for each bImportPkg do%>
//Imported packages
import <%path%>;
<%od%>

//class declaration
<%accessModifier%> <%fieldModifier%> <%name%>
  <%if .bAST.inheritanceExtends != "" %>
    extends <%inheritanceExtends%>
  <%fi%>
  <%for each bIInheritance do%>
    implements <%inheritanceImplements%>
  <%od%>
{

  //Attributes
  <%for each bAttribute do%>
    <%accessModifier%> <%fieldModifier%> <%type%> <%name%> = <%initialValue%>;
  <%od%>

  //Operations
  <%for each bOperation do%>
    <%accessModifier%> <%fieldModifier%> <%returnType%> <%name%>(
      <%for each bArgument do%>
        <%type%> <%name%>
      <%od%>
    ){
      <%body%>
    }
  <%od%>
}
<%od%>
```

3.7.2 Generated Java source code

As mentioned, we can generate a class from the previous template.An example of a class that we can generate is shown in Listing 3.22. Note that we replaced some source code by ....

Listing 3.22: Generated Java source code

```
package school.people;

//class declaration
public class Student {
  //Attributes
  public final static String STUDENT_FEMALE = "female";
  public final static String STUDENT_MALE = "male";
  private String name = "";
  private String gender = "";
```

```
  //Operations
  public Student(){
  }
  ...
  public synchronized String getName() {
    return this.name;
  }
  ...
  public synchronized String getGender() {
    if(this.gender == STUDENT_FEMALE){
      return new String("female");
    }else{
      return new String("male");
    }
  }
  ...
}
```

3.8 Verifying generated source code with the Java test case system

We validate the generated source code with a Java test case system. Please note that because of the limited implementation of the transformers in this project, we do this verification theoratically.

We will first present a Java test case, which we have developed. Then we discuss to what extend we can satisfy the requirements that are defined in Section 1.3.

3.8.1 Developing a Java test case system

We have developed a small interactive system for the verification of this project manually. We developed this test case manually, because if we would make a test case with modelling then we could never check whether the generated source code is correct or similar to manually developed one. So, the generated source code has to be developed independently from UML.

The test case system is a student administration system and it has a graphical user interface (GUI). The user can only add students to the system by means of the GUI which has three textfields and a button. The user will see two clients when (s)he starts the system. The server (our system) tries to handle the requests that the user gives.

Why the system is a good test case system

Instead of creating our own system we could also use an existing system as a test case system. In that case, we have to analyse the system and try to model it as it is developed initially. We have applied this alternative on a few systems, which are developed by other people. However, we cannot ensure that those systems contained all programming aspects that we would like to express. That is why we decided to develop a system of our own.

While developing this system, we made a list of different aspects that should appear in source code. The system consists of three classes, three inner classes, and twenty methods. We put different control structures (if-statements, while-loops, et cetera) to make the semantic part of a method (the body) more interesting to model. The system has 284 lines of code.

Because of the limited time we could not implement every aspect in the transformers. So, we selected the most important aspects of an object-oriented programming language. Aspects that we have selected are

inheritance, inner and nested classes, concrete and abstract classes, associations, global and local variables, for-loop, while-loop, switch-cases, and if-then-else.

Aspects that we have not selected are *exception handling*, *casting*, *arrays*, *data structures*. Note that we generate Java version 1.4 source code that conforms to the Java specification second edition. There is the aspect *parameterised type* that we should take into account. However, this issue only appear in Java version 5.0.

3.8.2 Comparing the generated code with the Java test case system

In this section, we discuss whether the generated source code satisfy the requirements (see Section 1.3). The requirements are (1) *the code must be well-formed*, (2) *the generated code must be semantically the same as the modelled behaviour*, and (3) *the architecture of the generated code should be the same as the code that is developed manually.*

Because the transformers and generator are implemented in ASF+SDf, we can ensure that requirement (1) is satisfied. The generation of source code is bound to the syntax of the target language. If the transformer tries to generate an expression that is not conform the syntax, the implementation of the transformer will not be parsable, i.e. not work.

The diagrams are modelled with a UML profile. We defined six rules in the UML profiles to which the diagrams have to conform to. We implemented the rules in the transformer (the merger of behaviour diagrams into class diagram). In this way we can ensure that the control structures are semantically the same as the modelled behaviour, which satisfies requirement (2). Note that we do not generate every control construct including `switch-case`, but we generate an `if-then-else` statement instead. However, the behaviour is still the same.

To satisfy requirement (3) we used the class diagram as the mapping with the architecture. The class diagram also shows the hierarchy of the packages that contain the classes and interfaces (when engineers choose to view the class or interface its fully qualified name). As a result, the class diagram contains enough information to make a good mapping with the architecture.

3.9 An overview of the transformers and generators in this project

An overview of several parts of our generator is shown in Figure 3.16. Note that this figure shows the architecture of our generator. The white box is a detailed view of the rectangle with the text *Generator of behavioural source code* shown in Figure 1.1 in Section 1.3.

As shown in the Figure 3.16, our generator takes the generated XMI of the UML tool as input. We transformed the generated XMI into our Interformat, because of the fact that the XMI is very tool-specific and contains many data that is not needed for code generation.

Then we merge the behaviour diagrams into a structure diagram (class diagram), because we wanted to simplify the generation process of source code. We transformed the behaviour diagrams into behavioural source code and put the results back into the Interformat document. After that, we removed the behavioural diagram from the tree. In the end, we have an Interformat document, that has the structure of a class diagram, but the behaviour is preserved.

In order to ensure that we generate well-formed Java source code without knowing the whole complex syntax of that language, we have chosen to use a template-based generator that is developed by [Arn06]. Therefore, we have to generate our Interformat into a BAST document.

The template-based generator generates (Java) source code by means of templates. We developed a template for classes and a template for interfaces.

In the end, we compared the generated source code with source code that is developed manually.

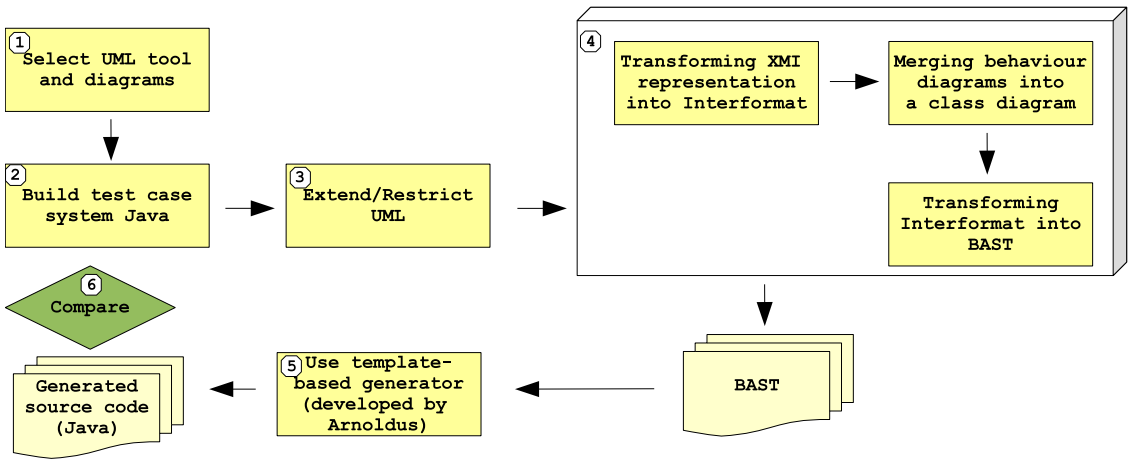


Figure 3.16: Visualisation of the project

Chapter 4

Results

In Section 1.2 we formulated our research question and four subquestions. In this Chapter we give answer to the research questions. The answers are discussed through the previous chapters including Chapter 3.

First, we will answer our research question. Then we will answer each subquestion apart. Our research question is

'How can behavioural code be generated using both structure and behaviour diagrams, despite the fact that UML does not provide a clear mapping between these types of diagrams?'

We have shown in Chapter 3 that it is possible to generate source code that not only contains the structure, but also the semantics of the system.

We defined a UML profile that contains stereotypes and restrictions. While a stereotype is actually a keyword that has a mapping with fragments of Java source code, the restrictions limit the modelling in UML diagrams. The restriction rules are needed to make automatic generation of source code possible.

We are able to use a structure diagram and one or more behaviour diagrams at the same time for code generation. This is possible because we defined a rule that restricts the naming of the behaviour diagrams. See Section 4.3 for the discussion.

We distinguish four levels of precision in UML, architecture, class, method, and attribute. We need to model the diagrams at the method and attribute level for generation of source code, because the distance to the source code is very little. The same goes for the class diagram, although this diagram has the class level.

We defined three requirements in Section 1.3. Those requirements were defined as follows: (1) *the code must be well-formed*, (2) *the generated code must be semantically the same as the modelled behaviour*, and (3) *the architecture of the generated code should be the same as the code that is developed manually.*

We have shown that our approach is able to fulfil all defined requirements. This because of the technique, restrictions, and the use of a class diagram in the generation process. The discussion of the requirements can be found in Section 3.8.2.

4.1 How are behaviour diagrams saved in XMI, and is that representation practical for code generation?

The results of this section are presented in Section 2.4, Section 3.3, and Appendix A.

How behaviour diagrams are saved in XMI In order to generate behavioural source code, we need to know how behaviour diagrams are saved. That is why we made a mapping between the UML elements of the activity diagram and XMI and the state diagram (see Section 2.4.1 and Section 2.4.2 respectively).

Both diagrams are saved in two parts. One part contains relations, while the other part contains the states or the actions of a diagram. These two parts are strongly connected to each other. Each part contains references to other XMI elements that are a part of the diagram. The exception on this rule is the submachine (element of state diagram), which has a reference to another state diagram.

There are two types of XMI elements, namely XMI elements that contains data (`ObjectStart`) and XMI elements that does not contain data (`ItemHdr`), but they indicate the type of the next XMI element. Every XMI element with data contains a unique identifier, which is indicated by `xmi.id`, while the references are indicated by `xmi.idref`. The sequence of the states and the activities can be found with the unique identifier of the references.

We concluded that the separation of data in the diagrams is good enough for code generation. It contains enough information in order to interpret the sequence of actions and states.

How practical an XMI is for code generation We analysed the XMI to gain insight in the XML-based format. The results are presented in Appendix A. We also discussed the XMI representation whether it is practical for code generation in Section 3.3.

We compared the XMI that is generated by Gentleware's Poseidon for UML with other UML tools. We noticed that the XMIs that are generated by UML tools is tool-specific. As a consequence, the diagrams that are created in one tool, cannot be interpreted by another tool. This also means that the generators that have these XMIs as input are also tool-specific.

An XMI file can have references to elements that appear in other documents. This makes code generation difficult from this format, because it has to solve these references first before it can generate code.

The XMI file that is generated by Poseidon is very verbose and it can easily expand to megabytes, since some XMI elements contain more data than needed and it contains data of the modelled diagrams' layout. We do not have to use the data of the layout in order to generate source code from the selected diagrams. So, the XMI file is unnecessary large for code generation. We also noticed that the larger the XMI file, the sooner a UML tool will cause an error. Note that this is actually a common problem for all tool-specific XMI versions.

Our conclusion is that XMI is not the best format to use for code generators, because (1) using this format makes the generator not reusable for other XMI files, (2) it can contain references to other documents, which are generated by other UML tools, (3) it is very verbose, and (4) it can expand to megabytes that leads to memory errors.

4.2 Which behaviour diagrams are suited for generating behavioural source code?

The results of this section are presented in Section 2.2, Section 3.1.1, and are based on literature [DLS+02, DSTW04, KNNZ00, RFW+04].

It is not needed to use all UML diagrams to make generation of behavioural source code possible. If we would use all diagrams for code generation, the generator would be unnecessary complex. So, during our research we investigated which diagrams we have to use in order to generate source code. The results were one structure diagram and two behaviour diagrams, namely *class diagram*, *state diagram*, and *activity diagram*.

The result of the literature study for this question is that they always use the class diagram and state diagram. A class diagram covers the static part of a system, while the state diagram covers the semantic

(behaviour) part.

A benefit of using the class diagram as our structure diagram is that there is a one to one mapping with the source code. A state diagram can also have a one to one mapping with the source code, but it depends on which level of precision it has been modelled. However, we noticed that if we only use a state diagram to model the semantic part, the generator that we were implementing should have knowledge of the system's context. Note that it is not possible to model the control flow of the objects with a state diagram.

We wanted to create a generator that is reusable in other context. So, we needed another diagram that can be used to model behaviour. After some investigation, we selected the activity diagram, because (1) this diagram shows the control flow of the actions and (2) it is possible to model this diagram on the lowest level of precision.

By using a limited set of the UML diagrams, we solve the following problems partly: *there is no one to one mapping between behaviour descriptions and the source code* and *behaviour does not always have to appear in source code as explicit statements.*

4.3 How can we constrain the modelling possibilities and simultaneously increase the expressiveness in behaviour diagrams?

The results of this section are presented in Section 3.2, and are based on literature [SW06, BRJ05].

UML is an open modelling language that allows engineers to add 'artefacts'. It also allows engineers to define stereotypes, tagged values, and restrictions, which is defined in a UML profile. As a result, we use a UML profile to constrain the modelling possibilities and to increase the expressiveness of UML.

Constraining the modelling possibilities Normally, restrictions in the UML profile is about restricting the elements in UML diagrams by means of the Object Constraint Language (OCL). The restrictions are expressed as free-form text. However, a normal generator will never be able to interpret the free-form text. Even with keywords it will be very hard to interpret, because what a generator will generate depends on the context.

That is why we defined restriction on a higher level. The generator does not have to take many modelling styles into account by restricting the usage of UML elements in a diagram. Note that we do not have the same possibility as OCL, because of the higher abstraction level and not using free-form text, which makes our diagrams less expressive. However, it is good enough for code generation.

We defined seven restrictions to which a model must conform to, but six of them are about restricting the modelling. The restrictions are defined as follow.

- *Each state must contain an action in the do-activity section.*
- *Every activity and state diagram has only one initial node/state and has one or more final node/state. Except for composite state, which has only one initial and one final state.*
- *An activity may have only one outgoing relation, but it may have a conditional relation to itself.*
- *A decision element contains one or more conditional relations and may contain zero or one relation without a condition.*
- *A conditional relation (a guard) can only be used after a decision element. It is not allowed to use a conditional relation between two activities.*
- *The sequence of a decision element (a shape of a diamond) must always be closed with a final element (a circle within a circle) or a merge element (also a shape of a diamond).*

The restrictions eliminate ambiguous interpretation of the diagrams. As a result of having this restrictions is that the implementation of the generator, which generates source code from the diagrams, is less complex, because the generator does not have to take every possibility into account.

Increasing the expressiveness in behaviour diagrams In addition to the restrictions, we also defined several stereotypes. We use a different notation than the official one that is defined in [BRJ05], because otherwise we will see the same stereotype appearing more than once in the XMI document (it would appear in the Interformat document also).

Our stereotypes have a fixed syntax (see Section 3.2.2). The stereotypes have a prefix notation followed by one or more arguments.

Our stereotypes have a clear mapping with Java source code. As a result we can generate fragments of source code. The generated code fragment will replace the old attribute value that contained the stereotype expression.

As a result of this UML profile, we eliminated the problems *there is no one to one mapping between behaviour descriptions and the source code*, *behaviour does not always have to appear in source code as explicit statements*, and *behaviour diagram can be implemented in different ways.*

4.4 How can we make the mapping between structure and behaviour diagrams clear?

The results of this section are presented in Section 3.2.

There are no rules about how the diagrams can be mapped together and every diagram can be used independently. Because of this missing, the mapping between the diagrams can only be done by engineers. However, the engineers will create a mapping or rules while they are modelling, but those rules are only in their mind.

So, we defined a rule that gives a clear mapping between the diagrams. The rule is defined as follows: *the diagram must be saved at the same level and have the same name as the related UML element.* It means that if the diagram is used to express the behaviour of a method, which is a leaf of a class hierarchy, it has to be saved as a leaf of the same class. In addition to the latter, the diagram must also have the same name as the related method, including the arguments of the method. The reason why the diagram must include the arguments of the method is because of the possibility of overloading in Java. In this way we will not generate behaviour source code for the wrong method.

As a result of solving this research subquestion, we also eliminated the problem *the mapping between structure and behaviour diagrams is not always clear.*

Chapter 5

Evaluation

In this chapter we evaluate the process steps that we have taken and the results.

5.1 Abstraction level in the diagrams

We have shown that it is possible to generate behavioural source code from UML diagrams by means of a UML profile. However, behaviour diagrams contained platform specific expressions. This way of modelling lies on the edge of the principles of visual programming. The question that remains is: 'would engineers model all methods on the same way like we did?' The answer would be 'no', because it is easier to add behaviour manually into the generated source code.

We are aware that the behaviour diagrams do not contain any abstraction, which is caused by the small set of stereotypes in the UML profile. We can solve this problem by adding new stereotypes and extend the resolver (Section 3.4).

Although we have platform specific expressions in our behaviour diagrams, they have no effect on the results that we have presented in Chapter 4. The stereotypes only hide the implementation details, but they not change the behaviour that is modelled in the diagrams.

Future work A question that rises is whether we can use the same stereotypes of our UML profile in a PIM. The stereotypes will probably have a syntax that is platform independent. As a result, the PIM-to-PSM transformer only has to transform the platform independent stereotypes into platform specific stereotypes.

5.2 The XMI and Interformat

XMI did increase the interchange of data between diagrams. However, its goal is still not achieved. As long as the UML tools do not save their XMI according to the official specification, engineers (who develop transformers) will transform the tool-specific XMI into an 'XMI light' format. This will make the own tool set less dependent on a specific MDA/UML tool.

The Interformat that we have developed is also an XMI light format. It is concise and well-formed, which makes it practical for the generation of code. Although we removed data of the XMI that we did not needed, it still can contain some data that is not useful. The latter does not have any effect on the presented results.

Future work The larger the system, the more diagrams are needed to specify the behaviour and its functionality. The more diagrams are needed, the larger the XMI. There is a need for a concise XML-based format that is suited for code generation. As a result, engineers do not have to transform the concise XML-based format into a lightweight format.

5.3 Generating control constructs

We are aware that we do not generate the most optimised source code. The generated code contains duplicate code. For example, a `while-loop` that is expressed in a diagram is generated into an `if-then` statement, which contains a `while-loop`.

Future work We think that we can solve this by making the analysis of the models better in our generator. We think that it is also possible to solve this problem by extending the UML profile with stereotypes or tagged values.

5.4 Checking the diagrams

We did not check the diagrams whether they were consistent before the generation of behavioural source code. The benefit of checking the diagrams is that errors between diagrams can be detected early in the generation process. As a result, the errors will not appear in the generated source code.

We did some research and developed a prototype for this, but because of the limited time we could not complete this activity. At this moment, the prototype is performing analysis on the Interformat and saves all of the data in a separate file. However, validation rules still have to be defined.

The prototype would have taken place between the steps discussed in Section 3.3 and Section 3.4 in our plan execution.

Although we did not have a checker for our diagrams, we checked them manually to make sure that the generated code does not contain errors.

Bibliography

[ABC+01] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible stylesheet language (xsl). Official standard specification REC-xsl-20011015, World Wide Web Consortium (W3C), 2001.

[Ano05] Anonymous. Objects by design, inc. Website, 2005.

[Arn06] J. Arnoldus. Balance abstract syntax tree and a template-based generator. Private conversation, 2006.

[Béz05] J. Bézivin. Model driven engineering: Principles, scope, deployment and applicability. preproceeding TR-CCTC/DI-35, Atlas Group, INRIA and LINA, University of Nantes, France, 2005. International Summer School 2005 on Generative and Transformational Techniques in Software Engineering.

[BK05] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual – Revision: 1.149*. CWI – Centrum voor Wiskunde en Informatica, 2005. `http://www.cwi.nl/projects/MetaEnv/meta/doc/manual.pdf`.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 2^{nd} edition, 2005.

[Cla99] J. Clark. Xsl transformations (xslt) version 1.0. Official standard specification REC-xslt-19991116, World Wide Web Consortium (W3C), 1999.

[DLS+02] E. Di Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, and M. Trombetta. Deriving executable process descriptions from uml. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 155–165, New York, NY, USA, 2002. ACM Press.

[DSTW04] K. Dang Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong. Model-driven soc design via executable uml to systemc. In *25^{th} IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 459–468. IEEE Computer Society, 2004.

[FS00] M. Fowler and K. Scott. *UML Beknopt*. Addison Wesley Longman, Inc, Zeist, The Netherlands, 2^{nd} edition, 2000.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating uml diagrams for production control systems. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 241–251, New York, NY, USA, 2000. ACM Press.

[LKT04] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. In J.-P. Tolvanen, J. Sprinkle, and M. Rossi, editors, *DSM'04: Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, Finland, 2004. University of Jyväskylä.

[Mod06] Modelbased.net. Mda tools. Website, 2006. `http://www.modelbased.net/mda_tools.html`.

[OMG02] OMG - Object Management Group. *XML Metadata Interchange (XMI) Specification version 1.2*, January 2002.

[OMG03] OMG - Object Management Group. *UML 1.5 Specification*, March 2003.

[OMG04] OMG - Object Management Group. *UML 2.0 Superstructure Specification*, October 2004.

[OMG05] OMG - Object Management Group. *MOF 2.0 / XMI Mapping Specification v2.1*, September 2005.

[RFW+04] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model driven architecture with executable UML*. Cambridge University Press, Cambridge, UK, 1 edition, November 2004.

[SW06] M. Staron and C. Wohlin. An industrial case study on the choice between language customization mechanisms. In J. Münch and M. Vierimaa, editors, *Profes 2006: Product-Focused Software Process Improvement*, pages 177–191. Springer-Verlag, 2006.

[WK01] J. Warmer and A. Kleppe. *Praktisch UML*. Addison Wesley Longman Nederland B.V. / Pearson Education Uitgeverij B.V., Amsterdam, The Netherlands, 2^{nd} edition, 2001.

Appendix A

State of the art: XMI

While analysing the XMI of several UML tools, we noticed that many tools differ in the way they save their model information. Even when they use the same XMI version. The differences are the names of the attributes, the names of the elements, or even whether they save the information as an element or as an attribute. This is very strange, because OMG made the XMI specification [OMG02] freely available on its site.

A.1 Errors in the XMI specification

We think that the inconsistency in the XMI specification caused the tool-specific XMI-versions. We have made a syntax definition from the EBNF specification. While making the syntax definition, we have discovered an ambiguity and some errors (references to other elements which do not exist any more). An example of wrong references is shown in Listing A.1. Rule number `9` has a reference to rule `8a`, but as the example shows, it has no `<8a:AttributeAsElmt>`.

Listing A.1: Wrong reference in EBNF specification of XMI

```
8.  <ObjectContents> ::= <8a:ContentElement> *
8a. <ContentElement> ::= <8b:ItemHdr>
                         <8c:ItemContent> *
                         <8d:ItemEnd> ?
8b. <ItemHdr>        ::= "<" <8k:ItemTag>
                         (8h:AttribEnumValue "/"? )?
                         ">"
...
9. <ClassAttributes> ::= ( <8a:AttributeAsElmt> )*
```

One of the ambiguities that we have discovered is shown in Listing A.2. Grammar rule `8c` can be one of the three following grammar rules `8e`, `8f`, or `8g`. If we put the focus on `8e` and `8g`, we can see that both grammar rules refer to `<7:ObjectAsElement>`. This reference causes an ambiguity in the syntax definition.

Listing A.2: Ambiguity in EBNF specification of XMI

```
8c. <ItemContent>       ::= <8e:AttribContent>
                            | <8f:ReferenceContent>
                            | <8g:CompositeContent>
...
8e. <AttribContent>     ::= <8i:AttribData> | <8j:AttribObject>
8f. <ReferenceContent>  ::= <7g:ObjectRef>
8g. <CompositeContent>  ::= <7:ObjectAsElement>
```

```
...
8i. <AttribData>          ::= // value // | "<" "XMI.reference" <2e:Link> "/>"
8j. <AttribObject>        ::= <7:ObjectAsElement>
```

Besides these errors we tried to parse the examples that is given by the specification. However, these examples are not parsable, because they are not well-formed according to the investigated XMI specification.

A.1.1 XMI specifications in DTD and EBNF

There is also a Document Type Definition (DTD). The difference between DTD and EBNF is that a language defined with EBNF is more abstract than when it is defined with DTD. As a result, the EBNF specification of XMI contains the most abstract syntax information (which is ten pages long), while the DTD contains the most concrete syntax information (which is one hundredandsixteen pages long).

A remarkable thing is that the DTD-specification is a representation of the UML 1.1 metamodel, while the currect UML version at that moment of writing that XMI was 1.3. Many UML tools use XMI version 1.2 to save UML diagrams of version 1.3 and later. As a consequence, the mapping between the XMI specification is not clear, which can also be a reason of all those tool-specific XMI versions.

A.1.2 The reasons of errors

We think that the developers of the UML tools were confused, because they did not know which definition is correct. The XMI examples that are given in the specification contain also errors that are not parsable, which can confuse the developers even more.

A second explanation is that UML tools already existed before the XMI specification, which has problems with XMI implementation as result.

A third explanation is that although companies want to use XMI, they do not want to make it conform the official XMI specification because of the competitors.

Appendix B

Architecture of Interformat and XMI

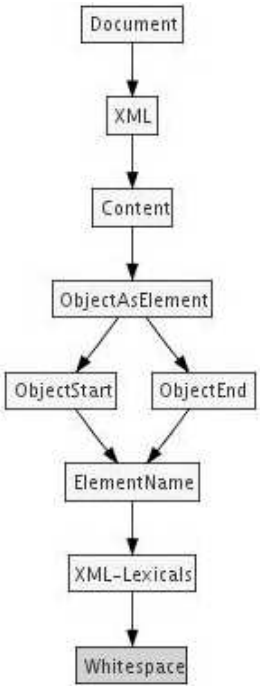


Figure B.1: Module view, architecture of interformat

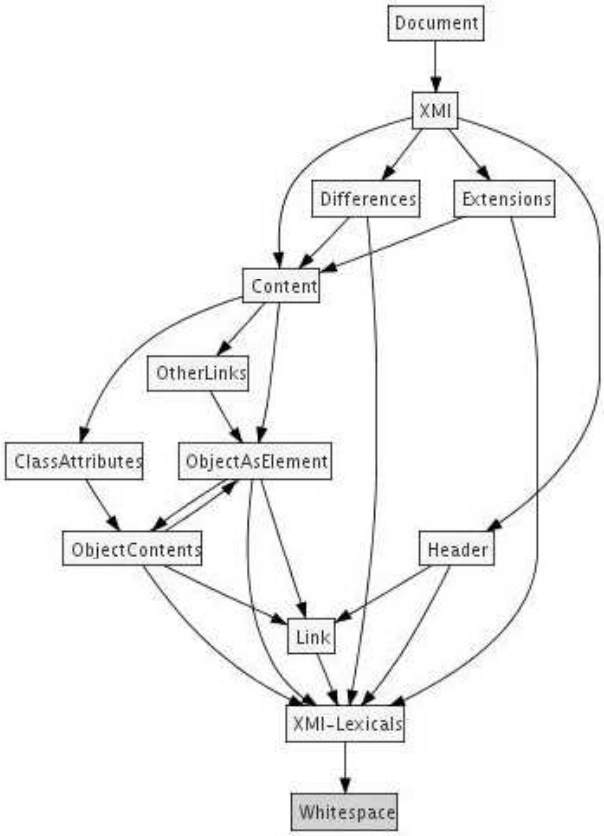


Figure B.2: Module view, architecture of XMI