MASTER

Automatic conversion of UML-class diagrams to POOSL

Friederichs, B.D.J.

Award date:
2004

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
AUTOMATIC CONVERSION OF UML-CLASS DIAGRAMS TO POOSL.

B.D.J. Friederichs

Supervisor: prof.dr.ir. R.H.J.M. Otten
Coach: dr.ir. P.H.A. v.d. Putten
Date: June 2004
Faculteitsbibliotheek Elektrotechniek
Postbus 90159
5600 RM Eindhoven
Tel: 040 247 25 32

Intern adres:
EH 2.06
Tel: 25 32

Dit werk uiterlijk terugbezorgen op laatst gestempelde datum

<table>
<thead>
<tr>
<th>2 november 2006</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

20991013
"There are fundamental limits upon a human’s ability to address complexity, and modeling is one of the most profound ways to help people reason, justify and construct systems of complexity."

— Grady Booch

This thesis is accompanied by a CD-ROM with documentation, code and online content as it was at the time of writing.
Foreword

Most people who have studied will acknowledge the fact that being a student is a great time. Some even say it was the best time in their lives. I can agree on the first, but not yet on the second, because I have a lot of life to live left. Being a student is great, which is the main reason I tried to lengthen that time as much as possible. I managed to be a student for eight years and I liked every bit of it. Okay, I must admit that sometimes there were courses that I didn't care much about, but later they came in useful for other courses or projects. Eight years I had the time and possibilities to extend my knowledge, and increase my skill. I realise that I am lucky that it was possible for me. But, all good things end and thus also my student hood. It is time to enter the real world and use all the knowledge and skill I am so happy to have acquired.

Those eight years culminated in a work, which took me nine months to complete. It is the work in front of you now. It is my master's thesis. During the course of those nine months I had the time to do research on a matter that is both interesting and confusing. Many times I needed to sit back, close my eyes and try to grasp the complexity of it or re-arrange things to void my confusion. It took me some months before I could explain the matter to fellow students and only shortly I managed to explain it to non-engineers. The concepts though, of class and object, of generalisations and associations, of structure and behaviour are all around us and that enabled me to place a lot in a wider perspective, although you will not find those ideas in this thesis. Maybe I will write something about that later. Then again, maybe I won't.

What you will find in this thesis is a research to manageability of complex real-time embedded systems. I tried to put existing methods together and automate conversion between them, using a tool I created. During the last weeks of the research interesting ideas emerged on how the future of that tool and the methods that I researched can evolve. It seems like the most interesting stuff always comes when you are almost finished. On the other hand, I think I needed the prior work to get to those discoveries. In this thesis I try to describe these ideas as well as possible, as well as the work done, obviously.

There some people I would like to thank. First off, Henk Friederichs, my father, for making those eight years of studying possible and of course the rest of my family and friends for moral support. During those last nine months, Piet van der Putten and Jeroen Voeten helped me in understanding the matter, especially during discussions about it. Also, Eric van Gerwen and Jinfeng Huang as fellow students and good discussion companions. For help during the programming of the tool, I received good help from the Java and XMI community, notably Pete Rivett of Adaptive Inc. and a lot of nameless people from around the world.

Bart Friederichs, Eindhoven 2004
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>Summary</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The assignment</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Contents</td>
<td>2</td>
</tr>
<tr>
<td>1.4 The need explained</td>
<td>2</td>
</tr>
<tr>
<td>1.4.1 Short overview of UML and POOSL</td>
<td>2</td>
</tr>
<tr>
<td>1.4.2 Bridging UML and POOSL</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Features of the converter</td>
<td>4</td>
</tr>
<tr>
<td>2 Modelling</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 The Meta Object Facility (MOF)</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 XML-based Meta-model Interchange (XMI)</td>
<td>8</td>
</tr>
<tr>
<td>2.2 POOSL</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Overview</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Metamodel and notation</td>
<td>9</td>
</tr>
<tr>
<td>2.3 UML</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Static structure</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Dynamic behaviour</td>
<td>13</td>
</tr>
<tr>
<td>3 POOSL – UML mapping</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Prerequisites</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1 Data types</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2 Stereotypes</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Static elements</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1 Classes and objects</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2 Interfaces and ports</td>
<td>17</td>
</tr>
<tr>
<td>3.4 Dynamic/behavioural elements</td>
<td>18</td>
</tr>
<tr>
<td>3.4.1 Concepts</td>
<td>18</td>
</tr>
<tr>
<td>3.4.2 Translation</td>
<td>19</td>
</tr>
<tr>
<td>3.4.3 Timing, abort and interrupt</td>
<td>21</td>
</tr>
<tr>
<td>4 The modelconvert tool</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Model authoring</td>
<td>23</td>
</tr>
<tr>
<td>4.1.1 UML</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 POOSL</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Usage</td>
<td>23</td>
</tr>
<tr>
<td>4.2.1 Creating the UML model</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2 Invocation</td>
<td>24</td>
</tr>
<tr>
<td>4.2.3 Typical use</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Design</td>
<td>27</td>
</tr>
<tr>
<td>4.3.1 Overview</td>
<td>27</td>
</tr>
<tr>
<td>4.3.2 Internal representation</td>
<td>27</td>
</tr>
<tr>
<td>4.3.3 XMI[UML] to POOSL conversion</td>
<td>28</td>
</tr>
<tr>
<td>4.3.4 POOSL to XMI[UML] conversion</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.4.1 Used technologies and tools</td>
<td>30</td>
</tr>
<tr>
<td>4.4.2 Tool class diagrams</td>
<td>33</td>
</tr>
<tr>
<td>4.4.3 Importing models</td>
<td>35</td>
</tr>
</tbody>
</table>
Summary

Ch. 1 Current developments in the real-time embedded systems field make it obligate to have tools and methods that can manage the complexity of it. More and more features are added to for example DVD players and mobile devices. Modelling languages have long been the tools for reaching that manageability. The software development field saw its languages converge into one unified language (Unified Modelling Language, or UML). Research to software/hardware co-design created the SHE method and an accompanying modelling language POOSL. This thesis describes the mapping between the UML and POOSL. Important factor of this project is the popularity boost of the POOSL language. During the course of the project the complete mapping was created as well as a software program that automates it partially. In the ending weeks of the research a meta-model of the POOSL language was created and a lot of future work was defined, including the UML to POOSL translation on a higher meta-level and the total convergence of the two languages.

Ch. 2.1 Modelling has always been a way for people to manage complexity and that has no way been different in technology. The Object Model Group (OMG) has defined a general language in which modelling languages can be defined. This is called the Meta Object Facility (MOF) and it is said to be a meta-meta-model. An instance of it is called a meta-model and the UML is a well known example of it. Besides the MOF, OMG also defined a ruleset to generate file formats to save models. This ruleset is called the XML-based Metamodel Interchange (XMI). It describes how to create XML Document Type Definitions (DTDs) or XML Schemas. For the UML such a XMI file format exists and is called XMI[UML].

Ch. 2.2 The Parallel Object Oriented Specification Language (POOSL) is a turing complete language that can be used to describe models of real-time embedded systems. It has asynchronous inter-object parallelism and synchronous rendez-vous communication between objects. There are also model creation and execution tools available, called SHESim (graphically drawing and executing models) and Rotalumis (High-speed or real-time execution of models). During the research a meta-model of the POOSL language has been created (figure 4).

Ch. 2.3 The UML is a graphical language mainly designed for software development and communication among engineers. It is object-oriented and uses different views to manage complexity. The newest version, 2.0, adds concepts needed for system modelling such as architecture diagrams and ports. For the most designs, only few diagrams (there are 13 in UML 2) are needed and for this research, the diagrams used are the class diagram, the architecture diagram and the state diagram.

Ch. 3 POOSL and UML have the same roots, and this makes it very well possible to create a consistent and correct mapping between the two languages. There are differences of course, and because POOSL is much smaller than UML, only the concepts that are not available in the UML need special attention. Things that are the same can be mapped directly and what is not available in POOSL can safely be ignored. The mapping is divided into three parts: prerequisites, static elements and dynamic elements. Prerequisites are the extensions needed in UML for modelling POOSL models. It uses correct UML to do so (stereotypes, packages). The static elements are mostly straightforward and use the prerequisites to reach their goal. This report aims at this mapping. The dynamic elements are included for completeness and are only guidelines and ideas for future work. They are state diagram centric.
Ch. 4  To automate static structure conversion between UML and POOSL (and back) a software program is developed, written in Java. It makes intensive use of XML technologies (such as XPath and DOM) to parse the XMI[UML] files into an internal UML model. After parsing, the tool converts the UML into an internal POOSL model, which is then written to SHESim or Rotalumis files. The other way round is also implemented, because round-trip engineering is important. It is even chosen to put two-way conversion above completeness. The tool can not convert the dynamic part of models; that is part of the future work.

Ch. 5  In conclusion, one could say that the UML to POOSL mapping is straightforward in a lot of ways and can therefore be automated very well. The shortcomings of UML can actually be solved with UML itself, thanks to its outstanding extension possibilities. The conversion actually makes it possible to execute UML models. Still, a lot of future work is still needed, both for the mapping, as for the tool and POOSL itself. The mapping and the tool need behaviour translation added. POOSL lacks one standard syntax to describe models and the tools (SHESim and Rotalumis) need to adhere to that syntax. In the far future, it might even be possible to converge UML and POOSL.
1 Introduction

1.1 Overview

Designing real time embedded systems has always been more an art than a formal design trajectory. Many systems are built bottom-up, starting with what was possible with a specific processor or what speed some bus had. With increasing complexity, the need has risen to keep that complexity manageable. Design had to be more top-down, starting with the customer's needs and wishes. Modelling languages are a very useful tool in this case, because they give the designer a platform-independent approach and make it easier to communicate with the customer or with fellow engineers.

The usage of modelling languages in the computer science field has been widely adopted for a fairly long time. Think of structured design languages such as defined by Hatley & Pirbhai [HP87] and Yourdon [Y89]. The last ten years more and more the shift went towards object-oriented design and languages such as Real-time Object Oriented Modelling (ROOM) [S94] and the Unified Modelling Language (UML) were developed [E98]. The embedded systems field, however, didn't have the privilege of such a design tool. There are some additions to existing methods and languages of the computer science field, but none of them really sufficed.

With the development of the Software/Hardware Engineering (SHE) method at the University of Technology of Eindhoven [vdPV97], and now lately, UML 2.0 [UML2.0], which has real-time additions, such modelling tools exist. The SHE method inhibits a modelling language, POOSL [POOSL], a simulation environment, SHEsim [G02] and a high speed execution engine, Rotalumis [vB02]. The new UML version contains architecture modelling possibilities and timing profiles. A lot of the existing problems can be solved when converging the SHE method and UML 2.

1.2 The assignment

The UML is more and more used in system design, both in education and in the industry. Model execution and checking, however, can not be done with the UML. The SHE method has the ability to execute and check models, with the POOSL language. Creation of a mapping from UML to POOSL will make it possible to execute the UML models in a POOSL context.

The assignment consisted of:

• Translate UML to POOSL:
  • Find the differences between both languages,
  • Create a mapping between the two languages,
  • Test the use of both UML and POOSL in the 'real world' (case study).

• The creation of a UML to POOSL conversion tool:
  • Do extensive research to existing tools,
  • Program a tool that is capable of future expansion to get round trip engineering.

The approach to creation of the mapping was formulated during the term. By using meta-modelling and existing modelling standards, the entire mapping fits in the framework in which UML was created too (the Meta Object Facility or MOF, more on that in paragraph 2.1.1).

The UML is very large any many models can be created with it. Not all those models can be mapped onto POOSL, only a subset of them. Figure 1 shows this schematically. Part of the assignment is definition of this subset. This is done by defining requirements to which the models must comply.
1 Introduction

Figure 1: Schematic view of the UML to POOSL mapping

1.3 Contents

This thesis describes the research needed for the UML to POOSL conversion tool and actual creation of it. It is divided into two parts. Chapter 2 and 3 describe the research done and how the mapping to POOSL is done. The specifics of the languages are described there; where the differences appear in their syntax and semantics, if present. It starts with a general introduction in modelling and important concepts used. The POOSL paragraph (2.2) shows the meta-model, created during the research including its mapping to the textual representation of the language. Paragraph 2.3 shortly describes the UML. Only parts that are used in the mapping are included. The separate modelling languages are then converging in chapter 3, where the UML to POOSL mapping is laid out completely.

The second part of the thesis describes the development of the conversion tool in chapter 4. This is the more practical chapter of this thesis and is divided in four parts: creation of UML/POOSL models (4.1), usage of the conversion tool (4.2), design of the tool (4.3), and its implementation (4.4). The part on the implementation also has an overview on the used technologies in the tool. Finally, chapter 5 describes all conclusions and recommendations on future work on this project. Throughout the thesis, parts of future work can be found. They can be recognised by the annotation 'FW' printed in the margin.

The first appendix, A, has an exhausting list of all abbreviations used throughout the thesis and where to find its description. Appendix B shows a research to existing UML authoring tools, and how they make use of existing standards. Appendix C describes in detail the inner structure of XMI[UML]. Appendix D explains the typical use of the tool by example of a case study.

1.4 The need explained

1.4.1 Short overview of UML and POOSL

The modelling method Software-Hardware Engineering (SHE) is developed at the Technical University of Eindhoven. This method features real time, object oriented modelling. [vdPV97] is the base work of the SHE method and in it, the most important existing modelling languages are discussed and how they relate to software/hardware engineering; see also [vG04]. SHE is based on concepts from those languages and was its time ahead, as it implements concepts that are only to be found in the latest version of UML.

The UML is based on the same languages as SHE: in 1994, some of the people1 of different modelling languages decided to work together and specify a unified language, which later (in

---

1 To be precise: Grady Booch (the Booch method) and James Rumbaugh (OMT) of Rational Rose. In 1995 Ivar Jacobson (OOSE/Objectory) joined them. They were called 'The Three Amigos' later on.
1997) became the Unified Modelling Language (UML). This ended what was called 'the method war'. Currently the UML is in version 2.0 and since that version it inhibits elements to model architecture. The last version isn't widely used yet. The best-known version of the UML is 1.3. Some notable points on the usage of the UML in general:

- UML is widely accepted as a modelling language, throughout the entire software development field,
- UML consolidates years of research into one language. It is well thought-out and is the de facto standard of modelling languages,
- The use of different views makes the complexity manageable.

The following time line shows how several modelling languages have influenced others.

![Time line of object oriented modelling languages](image)

More and more tools are being developed to model in UML and run those models. This increases the popularity of UML in the software design field and with the coming of UML 2.0, this will also be the case for the embedded field, because timing and architecture are part of the new UML version.

Together with SHE a language was developed, POOSL (Parallel Object Oriented Structured Language). This language has all the constructs and semantics to build and run the models defined in SHE. POOSL has been evolving since its birth in 1995 [V95] and now the following tools and features are available:

- A graphical simulator called SHESim,
- A high-speed and real-time\(^2\) execution engine, Rotalumis,
- A delay construction to define the passing of virtual time,
- Rendezvous communication between objects.

More information about POOSL can be found in 2.2. A complete description of SHE and POOSL can be found in [vdPV97], [G02] and [vB02].

1.4.2 Bridging UML and POOSL

As explained before, both UML and POOSL are important in the real-time embedded design field. UML mostly being used for communicating with other engineers and POOSL being used for building executable models of systems. A tool that converts models between the two languages creates a platform on which complete systems can be built from whiteboard design to implementation. Figure 3 shows a typical design cycle from requirement to implementation and the used tool beside the phase. The mapping from POOSL to C++, has been researched and defined in [vB02]. Tools exist that can synthesize C++ to ICs, closing the gap to hardware.

---

\(^2\) Rotalumis comes in two versions: standard, which is high-speed and real-time, which runs in real-time and can control devices.
The UML standard as released by the OMG has no execution semantics defined. The OMG has issued a Request For Proposal (RFP) for the definition of action semantics. The Action Semantics Consortium is a group of companies (Telelogic, Rational, among others) that has submitted a response to that request, which was adopted November, 20th, 2001. See [ASC] for more information. The UML action semantics are released as a separate standard now by the OMG.

Some companies also have developed their own way of executing models, but they lack the formal semantics present in POOSL. When there is a way to translate the UML models into POOSL models, that would add semantics to the models, and also tool support for actually executing them. Another reason for this conversion is that it can boost the popularity of POOSL and the SHE method in the industry.

1.5 Features of the converter

The converter, as described in chapter 4, has several features that are important in its design.

Meta model independent design

The converter is designed with extensibility in mind. To have a tool that converts UML to POOSL one way wouldn't be very useful. Back annotation (using simulated results back in the modelled system) and round-trip engineering (UML to POOSL, back to UML) are very important in the design cycles of an embedded system. You never get it right the first time. An important feature of the converter therefore is round trip engineering, having a higher priority than completeness. Rather have a tool that does two-way conversion of a limited part of the model than one that does full conversion, but just one-way.

This leads to a tool that can easily be extended to other (modelling) languages and conversions between them. A future enhancement of the tool, could be the building of a plug-in system to make it easier to extend the tool.

Platform independence

Most tools used in the SHE method are platform independent or there are several versions of the tool for several platforms. Because the model converter tool is written in Java, it is also platform independent. It introduces some other benefits, such as easier design and extensive
support for XML processing, which is important because the standard UML file format is an XML file (more on this in 2.1.2).

**Standards compliance**

The tool uses well-known standards for in- and output of models. For UML models it uses XMI [UML] (XML-based Metadata Interchange, applied to the UML meta-model), version 1.2 as defined by the Object Model Group (OMG). The generated POOSL can either be fed to Rotalumis or SHESim (depending on option switches). Internally it uses XML standards such as DOM and XPath.

**Model features**

The model converter aims to do a conversion as extensive as possible. Currently, the tool only converts class information. In that aspect it supports conversion of the following POOSL features:

- process classes
  - methods
    - input parameters
    - output parameters
  - initial method call
  - attributes
  - instantiation parameters
- data classes
  - methods
    - input parameters
    - return value
  - attributes
- generalisations (both for process and data classes)
- primitive data classes (Real, Integer, Boolean and Char)
- POOSL to UML conversion (only SHESim syntax input)
- UML to POOSL conversion (both SHESim and Rotalumis output)
- round trip engineering (by re-using original diagram information)

---

3 See chapter 2.2 for a complete description of the concepts
2 Modelling

2.1 Introduction

2.1.1 The Meta Object Facility (MOF)

The Object Management Group (OMG) has described a standard called the Meta Object Facility (MOF; [MOFIA]) that creates a framework in which meta-models can be created. Meta-models are the complete description of what is valid within a modelling language and also describe the semantics of that specific language. One could say that a meta-model is a model of a modelling language. The MOF goes a step further and defines a model that can be used to create meta-models, which makes it a meta-meta model. In this perspective, you could say that:

- data is an instance of
- a UML model, which is an instance of
- the UML, which is an instance of
- the MOF

See table 1 for the four-level MOF framework, with UML and POOSL as examples in it.

Table 1: Four layer MOF framework

<table>
<thead>
<tr>
<th>MOF</th>
<th>UML</th>
<th>M4 (meta-meta model)</th>
<th>M3 (meta model)</th>
<th>M2 (model, meta data)</th>
<th>M1 (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOSL</td>
<td>UML model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POOSL model</td>
<td>UML model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POOSL runtime objects</td>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the UML, a meta model is available [UML2.0], defined by the OMG. POOSL does not have such a model, and in the course of this research, the first version was created. See paragraph 2.2.2 for the meta-model and an explanation. The MOF meta-model of POOSL is created to make the conversion simpler and more consistent. When both meta-models (POOSL and UML) are described within the same framework, it is much easier to translate between them. Doing the translation on the M3 meta level will create a very generic translation, which can be extended to other tools.

The conversion on the M3 meta level is very abstract and is not included in this thesis. The UML to POOSL translation is only made on the M2 meta level:

- Doing it on the M3 level now is too hard (more extensive knowledge of MOF is needed). To put this level of complexity in respect: the M3 mapping should be done in the course of a PhD project.
- It creates a readily usable tool and 'paper' mapping
- Because of the pragmatism, problems arise earlier and clearer, even when only some rudimentary knowledge of POOSL and/or UML is available

Chapter 3 describes the entire mapping, future work includes extending the mapping to the M3 meta level.

---

4 This is a relative level, the whole `meta` concept should be thought of as relative (unless you want it to drive you mad). See the MOF specification for more on this.
2 Modelling

2.1.2 XML-based Meta-model Interchange (XMI)

Besides the MOF, OMG also defines the XML-based Meta-model Interchange (XMI), which is a set of rules to create an XML-based file format for any MOF-based meta-model. This includes the following:

- Rules for creating DTDs (version 1.x) or XML Schema (version 2.0)
- Rules for creating XML files, so that they comply with the DTD/Schema and the XMI (mere DTD/Schema compliance is not enough)

It is important to understand that XMI is not a file format itself, it only defines rules by which file formats can be created. Per MOF-compliant meta-model, a set of rules can be created, that comply with the XMI. That set of rules then defines the file format, released as a DTD or an XML Schema. The name of the file format will be XMI followed by the name of the meta-model in square brackets, for example XMI[UML]. For extra clarity, it is possible to add version numbers to that, which then gives for example XMI1.2[UML1.4]. This thesis uses the terms XMI1.2[UML1.4] and XMI[UML] interchangeably. More information on XMI can be found on the OMG website and in the XMI standard [XMI1.2].

2.2 POOSL

2.2.1 Overview

Parallel Object Oriented Specification Language (POOSL) is a Turing-complete, object oriented language, mainly used for designing embedded real-time systems. It was developed at the Eindhoven University of Technology.

Its main features are:

- process-centric modelling
- asynchronous inter-object parallelism
- synchronous rendez-vous communication
- non-deterministic behaviour
- usage of virtual time for time passing
- full object oriented programming
- possibility the guard statements with expressions

Some of the features are discussed below.

Object orientation

POOSL introduces three different types of classes to build a system: cluster classes, process classes and data classes. Data classes are the most similar to normal classes, with the limitation that multiple inheritance is not supported.

Process classes differ strongly from the standard concept of a class. They have no constructors, cannot be abstract and also don't support multiple inheritance. Their instantiation is done in the creation of the architecture; they are not dynamically instantiable. Cluster classes differ even more, they have no attributes, no methods and do not support inheritance. Cluster classes are used to create hierarchical models.

Both data and process classes can have methods and attributes, which are called "instance
variables”. Process classes can also have instantiation parameters and an initial method call, that is called when the process starts. Cluster classes can only have instantiation parameters that can be used in the process class objects that are defined inside it.

**Parallelism**

POOSL supports two types of parallelism: between process objects and inside process objects. The parallelism between process objects means that separate process objects have their own runtime environment. Synchronisation with other objects can be done by synchronous, rendezvous communication. Think of separate processes in a multi-tasking system as an analogy. Parallelism inside objects means that one process object can execute several process threads, within the same runtime environment.

**Non determinism/real time behaviour**

POOSL was designed as a language to model real-time systems. This is the reason it introduces non determinism; it is possible to make a non deterministic choice of several options.

POOSL has real time capabilities that enables the user to use time as a resource. To do this in a sane way, POOSL uses virtual time that must be explicitly advanced by the modeller, by using the delay primitive [G96].

### 2.2.2 Metamodel and notation

On the next page, figure 4, is the MOF meta-model of the POOSL language. The dynamic behaviour is not modelled yet on this level. Only the static structure is defined here.

<table>
<thead>
<tr>
<th>Where in the MOF are we?</th>
<th></th>
<th><strong>Meta level</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOF</td>
<td></td>
<td>M4 (meta-meta model)</td>
</tr>
<tr>
<td>POOSL</td>
<td>UML</td>
<td>M3 (meta model)</td>
</tr>
<tr>
<td>POOSL model</td>
<td>UML model</td>
<td>M2 (model, meta data)</td>
</tr>
<tr>
<td>POOSL runtime objects</td>
<td>data</td>
<td>M1 (data)</td>
</tr>
</tbody>
</table>

This paragraph shows the mapping from the POOSL meta-model to the existing textual representation as used by Rotalumis. It is a blueprint M3 to M2 meta level mapping. Actually, it is not a real M3-M2 mapping, but when it is applied to an actual MOF[POOSL] model (an instance of the POOSL meta-model, given graphically), a mapping emerges. You could say it is a meta-mapping, but to prevent confusion it is said to be an M3 to M2 mapping.
All the non-abstract instantiable elements also have a textual notation, that is used in tools as Rotalumis and SHESim. The following list shows all these notations, in a slightly modified Extended Backus-Naur Form (EBNF):

- *italic print* refers to the corresponding element in the meta-model, to be included literally
- *function[]* defines an EBNF function, to be applied to the identifier from the meta-model between the brackets
- *whole.part* refers to a part of a referred element
- *regular expressions are allowed*

Any other constructs (quotes, parentheses, asterisks, etc) follow the EBNF standard.

Top Level Specification := 'system specification' name

  'behaviour specification (' procclu_list ')'\ (' channel_list ')'`
  ProcessClass`
  DataClass`
  ClusterClass`

ProcessClass :=  
  'process class' name '(' die_list[instantiation parameter] ')'`
  ('extends' superclass.name)?
  'instance variables' die_list[instance variable]
  'initial method call' pm_ref[initial method call]
  'instance methods' pm*

die_list :=
  (die ('; die)*)`

die :=
  identifier ': name`

identifier :=
  [A-Za-z][A-Za-z0-9]*`

pm_ref :=
  name '()()'

pm :=
  name '(' die_list[input parameter] ')' die_list[output parameter] ')'`
  (' | die_list[local variable] ')?
  code ':`

DataClass :=
  'data class' name (extends' superclass.name)?
  'instance variables' die_list[instance variable]
  'instance methods' dm*

dm :=
  name '(' die_list[input parameter] ')' return value.name
  (' | die_list[local variable] ')'?
  code ':`

---

6 Normal EBNF can not be used, because it only describes syntax and it is not possible to refer to a meta-model.
7 In the notation, some names are abbreviated: die – DynamicallyInstantiableElement, procclu – Process or Cluster, pm – Process Method, dm – Data Method
2 Modelling

ClusterClass := 'cluster class' name '(' die_list[instantiation parameter] ')' 'behaviour specification (' procclu_list ')' channel_list ')' procclu_list := procclu (' | ' procclu)* procclu := identifier ':' proccluclass.name '[' channel.name '/' proccluclass.port.name ']' channel_list := channel.name (' , ' channel.name)*

It is chosen to use the Rotalumis syntax throughout this document, because:

- The Rotalumis syntax is cleaner,
- The Rotalumis syntax is the latest standard,
- The SHESim syntax contains errors.

See table 4 for a complete overview of the syntactic differences between Rotalumis and SHESim.

The graphical (MOF) and textual models both define the exact same meta-model. They are included here both, because the textual representation already existed and needed a mapping to the MOF meta-model. Also, it serves as a blueprint for POOSL models.

Future work in this field includes:

- Refining the meta-model (including the behaviour),
- Checking it against the MOF,

2.3 UML

2.3.1 Introduction

The Unified Modelling Language (UML) is a graphical language that is widely used in the software design field. It aims to comprise everything in an object-oriented matter. UML can be considered the definition of object oriented design, which means every object oriented concept is included. A small, and certainly not complete overview is given here. Basic knowledge about object oriented concepts is assumed available.

UML uses several diagrams to show several 'views' of a system. This makes the complexity maintainable by ignoring parts that aren't useful in a particular view. For example, a view that describes how a user will interact with the system has no need for information about internal data structures. UML 2.0 has thirteen different diagrams. Only the ones useful for the UML to POOSL mapping are described in the following paragraphs. See chapter 3 for examples of UML, related to POOSL.

As said, UML is a graphical language. All elements are drawn by lines, rectangles and arrows, among others. See chapter 3 for examples. For more in-depth information about UML, refer to [UML2.0] for the specification or [E98] for a good starting guide.

Almost all designs can be divided into two parts: static structure and dynamic behaviour. The static structure usually describes internal data structures and how classes/objects communicate to each other. The dynamics describe how the classes will behave. This division is used in this thesis, starting in the following short introduction to the UML.
2.3.2 Static structure

The main diagrams used in the static structure are the class diagram, object/deployment diagram and communication diagram (collaboration diagram in UML 1.x). The class diagrams show how the classes are built and how classes relate to each other. Things such as methods, attributes, generalisations and associations are described in these diagrams. When building a software program, this usually is the diagram used most often and also where developers start their design.

In the embedded field the object/deployment and communication diagrams have more value, because those describe actual instantiations of those classes. In the embedded field, the amount of object instances per class is usually limited to a small number (tens or hundreds of them), whereas in e.g. database programs the instances can go up to the millions. The object/deployment diagram describes what instances of classes are present in a system. The communication diagram describes what messages are sent, as well as the order in which they are sent. When the embedded developer has defined these issues, he or she will start thinking about what classes to define, based on the objects that are in the design.

Earlier versions of the UML (1.x) weren't able to describe certain parts of an embedded design. Especially architecture was hard or even impossible to model correctly. Many of those problems are solved in the last version, 2.0. It is possible to give classes a 'port' through which they can communicate. This enables the current embedded designers to also model the architecture of an embedded system.

2.3.3 Dynamic behaviour

Diagrams to model behaviour are sequence, state and activity diagrams. Each of those have different goals. In the sequence diagrams, the order in which messages are sent to different objects is defined. Objects can be created and destroyed in this kind of diagram. It is often used in the design of information systems (software development). State diagrams describe the states in which a certain class can be. There are no instances of classes in the diagram, but they are linked to classes. These diagrams are useful for modelling the control of processes. Activity diagrams finally describe the different activities a system can do. It has no explicit objects but it can be defined what classes do what activity.

For the conversion to POOSL it is chosen to only use state diagrams because the conversion is class-centric, and so is the state diagram. OMG defined UML profiles and model elements for modelling time [OMG03]. This profile defines different concepts in the Time Model, whereas POOSL only has the delay primitive to advance time. How the concepts of the UML profile should be mapped onto POOSL is not researched.
3 POOSL – UML mapping

3.1 Introduction

When considering the differences between POOSL and UML, only the concepts that are not supported in UML need to be looked at. The concepts in UML, but not in POOSL, can easily be discarded by just not modelling them. This is because the UML is vastly larger than POOSL.

An example of a concept not available in UML is the existence of process classes in POOSL. All POOSL models must have at least one. A designer must take that into account and can only create models that are part of the subset that can be converted to POOSL. UML models that are not part of that subset need modification to get into it. This is not possible for all UML models.

In the following paragraphs there is an exhaustive list of all POOSL concepts and their UML 2.0 counterparts. To describe the POOSL constructs it is chosen to use generic examples.

The list is divided into three parts:

- Prerequisites: Before UML can be used as a modelling language for POOSL, it needs some constructs that are used in the UML modelling. Those elements are all valid UML 2.0.
- Static elements: Here the static elements of a POOSL model are mapped onto UML.
- Dynamic/behavioural elements: This is where the dynamic (class instantiations, among others) and behavioural elements are defined.

<table>
<thead>
<tr>
<th>Where in the MOF are we?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOF</td>
</tr>
<tr>
<td>POOSL</td>
</tr>
<tr>
<td>POOSL model</td>
</tr>
<tr>
<td>POOSL runtime objects</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

This chapter describes the UML to POOSL mapping on the M2 level.

3.2 Prerequisites

3.2.1 Data types

Data types can be considered as being primitive classes. Sometimes they are also called primitive data types or just primitives. Most programming languages have support for primitive data types, even an 'everything-is-an-object' type of language such as Java. POOSL is a non-typed language and uses primitive data classes. They are: Integer, Char, Real and Boolean [vdPV97].

UML has support for data types. They are defined as a specialisation of classifier ([UML2.0], page 94). This is a meta type and has no instantiations, such as Integer or Char in the UML specification itself. The POOSL meta model does define the primitive data classes used and its UML counterpart will be a package named “POOSL”, which holds four UML data types, Boolean, Char, Integer and Real. In the strict sense this is incorrect, because they should be classes instead of data types, but it keeps the conversion simpler.
3.2.2 Stereotypes

A stereotype is a UML extension mechanism. It enables the use of platform specific terminology and can be applied to all UML elements ([UML2.0], page 580). Stereotypes are plain text strings and are shown between guillemets ("<<" and ">>"), near (usually before the name of the element) or inside the UML element it applies to. Because UML 2.0 has no precise constructs needed for defining POOSL models, it is extended. The following stereotypes must be defined before a UML model can be correctly translated to POOSL:

Table 2: Required stereotypes for POOSL modelling

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;cluster&gt;&gt;</td>
<td>classifier</td>
<td>Defines a class as a cluster class</td>
</tr>
<tr>
<td>&lt;&lt;data&gt;&gt;</td>
<td>classifier</td>
<td>Defines a class as a POOSL data class</td>
</tr>
<tr>
<td>&lt;&lt;initial&gt;&gt;</td>
<td>operation</td>
<td>Defines a method as the initial method call</td>
</tr>
<tr>
<td>&lt;&lt;instantiation&gt;&gt;</td>
<td>attribute</td>
<td>Defines an attribute as an instantiation parameter</td>
</tr>
<tr>
<td>&lt;&lt;interface&gt;&gt;</td>
<td>classifier</td>
<td>Defines a set of messages</td>
</tr>
<tr>
<td>&lt;&lt;process&gt;&gt;</td>
<td>classifier</td>
<td>Defines a class as a POOSL process class</td>
</tr>
</tbody>
</table>

3.3 Static elements

3.3.1 Classes and objects

Classes and objects are the base elements of every object-oriented design. Because there is a large history in object-oriented modelling and UML and POOSL share some of this history, most concepts can be mapped pretty straightforward. The mapping will be given using generic examples, because it is defined at the M2 meta-level (see also chapter 2.1). On the left, a generic UML element is shown, and its POOSL counterpart is given at the right.

Process class

```
<< process >>
extends superclass
instance variables
var2: type
initial method call method1
instance methods
method1(inparam: type)(outparam: type)
code.
method2(inparam: type)(outparam: type)
code.
```
### Data class

Data class name
extends superclass
instance variables
type
instance methods
method (inparam: type): type
code.

### Cluster class

Cluster class name (var1: type)
behaviour specification
(name1: preprocess [inchannel/ inpre, internchannel/ outpre] || name2: postprocess [internchannel/ inpost, outchannel/ outpost]) \{internchannel\} [input/ inchannel, output/ outchannel]

**General notes:**

- Visibility has no meaning in POOSL and is therefore ignored. The default UML visibility used here is `public`, because in POOSL everything is public.
- Parameter lists are comma separated in Rotalumis
- A top level specification (system) is modelled just like the architecture model of a cluster class

#### 3.3.2 Interfaces and ports

UML 2.0 introduces the use of ports for all classifiers ([UML2.0], page 167). This is very helpful in the mapping to POOSL, because POOSL process classes also can have ports. As a matter of fact, a process class without ports is not very useful. Ports are defined by drawing a small rectangle on one of the edges of the classifier. A name must be placed near this rectangle. Furthermore must every port provide and/or require one or more interfaces. The name of the interface is printed near the end of the interface. Provided interfaces are drawn as a filled dot, required interfaces as a semi-circle. See figure 5 for an example.
There is no direct translation possible from UML to POOSL, considering the ports and messages. The modelling of the ports and the interfaces is mostly for documentation. In the future, the mapping might be useful. Think of an extension of the POOSL language in which interfaces are defined. A POOSL parser or compiler can then check all communications and ports against those interfaces and report an error when something is used incorrectly. This would increase the ease of which engineers can work together; interfaces must be defined and are also forced to comply.

3.4 Dynamic/behavioural elements

The main aim of this research was getting a mapping from UML to POOSL, limited to structure (system/clusters) and static elements (data/process classes). However, a system is nothing without behaviour. This paragraph shows some ideas on how behaviour can be modelled in UML and then translated into POOSL. The ideas aren't checked for correctness yet. Future work is aimed at extending and optimising the behaviour mapping.

The most obvious way to model behaviour is the use of state diagrams. UML has good support for those and it is pretty straightforward to translate them to POOSL. In the course of the research no real work was done towards this, but some ideas emerged making a case study.

Those ideas are presented here only for completeness. It serves as a starting point for future work. No behaviour modelling is available in the given meta-model of chapter 2.2.2, the mapping is done entirely on the M2 meta-level.

3.4.1 Concepts

State machines describe the behaviour of process classes and therefore are linked to them. Some concepts are important to understand, table 3 shows the concepts, what they mean and how they can be mapped to POOSL.

---

8 SHESim describes them explicitly in the source code, but this introduces redundancy, which is not a good thing in a formal language.
### Table 3: State machine concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description (UML syntax)</th>
<th>POOSL mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>This is where every state machine starts its execution. From this state there is only 1 transition and it is taken always (drawn as a filled dot in UML).</td>
<td>This is the 'initial method call' concept in POOSL, which can be seen as a reference to a method (which is a state, see further).</td>
</tr>
</tbody>
</table>
| Transition    | How and when a process will go on to another state. Three types of transitions exist:  
  - Unconditional transition – taken always,  
  - Conditional transition – only taken when the condition for it is true,  
  - Multiple transition – only transitions of which the condition is true can be taken. When more than one condition is true, one transition is taken non-deterministically. (drawn in several different ways, see 3.4.2). | Can be done using either the sel, par or if constructs of POOSL. The actual transition is a mere function call. Note that this makes use of the tail recursion of POOSL and this might introduce model errors. After a tail recursive method call, no other executable statement may occur. Otherwise, the internal stack of the POOSL execution engine builds up and might cause a crash (due to a stack overflow) at a later time. Transition can be guarded by expression guards or communication actions. |
| State         | A user-defined state in which the object is (drawn as a round-edged rectangle in UML).                                                                                                                                                                                                                                                                  | In this mapping, it is chosen to model them as methods with the name of the state.                                                                                                                                                        |
| Activity      | A set of actions, that are executed when an object is in a specific state. Activities can be guarded. Three activities are always in a state: entry (executed after entry of the state), do (executed while inside the state) and exit (executed before leaving the state).                                                                                                           | This is modelled as the code that is inside the state-method.                                                                                                                                                                               |
| Action        | An atomic operation, part of an activity. Can also be placed on a transition to create a Mealy-machine.                                                                                                                                                                                                                                                 | Modelled as code before the transition is taken (i.e. the state method is called).                                                                                                                                                         |
| Final state   | This is where the state machine ends (drawn as a filled dot, surrounded by a circle in UML).                                                                                                                                                                                                                                                                  | Not modelled. The final state is reached when there is no statement left. This means the process class is in deadlock. It shouldn't happen.                                                                                                   |

#### 3.4.2 Translation

As described above, states are modelled as methods. This paragraph will show the translation of the concepts from UML to POOSL. The translation back is not implemented because creating state machines from code is extremely complex and can be done in many ways. Sometimes even, it is impossible to create state machines and other means are necessary for modelling the code. Just copying the code verbatim into the methods will be sufficient for the round-trip engineering (the tool supports this).
Initial method call

process class name()
instance variables
initial method call firstState()
instance methods

Simple transition

Note: The guard expression $E$ can be empty. This is true for the $E$ expressions in the other transitions as well.

Multiple transition

Note: When there is a state after B, called B', use the simple transition syntax. The B' state is then called from the B state. The B' state should not call a next state in POOSL. Execution continues in the par construct, in the A state.
If transition

Note: The true and false branches can be connected anywhere on the diamond, as long as they are labelled clearly.

Activity

Notes:
- The S statement is POOSL code, as defined in [POOSL].
- In composition of several constructs (transitions and actions), state transitions are taken after the S statement is executed. This is modelled in POOSL by putting S before any state transition code.

The choice for some of these constructs introduces semantics not present in the UML. The mapping to POOSL adds them.

3.4.3 Timing, abort and interrupt

Besides the trivial transitions given in the previous paragraphs, there are some concepts that are available in POOSL, but that have no direct translation to UML 2. Some first ideas on how to model those are given here. These ideas must be validated both for POOSL and UML correctness. Here also, stereotypes or tags can be used to refine the semantics.

The three concepts given here are:
- Timing: POOSL has has real-time concepts since 1996 [G96]. It is implemented as a primitive delay.
- Abort: POOSL can abort a running statement with another statement. Execution of the aborted statement then ends.
- Interrupt: a statement can be interrupted by another statement. The first statement continues execution after the second has finished.

Timing

Timing can be added in various ways in a state machines:
- As a 'guard' on a transition edge, for example for a time out.
- Inside a statement compartment of state.
Abort

UML 2 introduces state entry- and exit points. The exit points can be used for the abortion of states.

\[
\text{A} \rightarrow \text{B} \quad \text{with} \quad \text{abort} \\
\text{A}()() \quad \text{with} \quad \{E\} \text{B}()() 
\]

Interrupt

The interrupt concept has some resemblance to the abort concept, so it could be modelled the same way in UML. However, UML also defines an interrupt. For completeness, both are given here. More research is needed to figure out if there is any semantic difference between the two and if so, choose the one that is closest to POOSL.

Option 1: Model it with a state-exit point, labelled 'interrupt'

\[
\text{A} \rightarrow \text{B} \quad \text{with} \quad \text{interrupt} \\
\text{A}()() \quad \text{with} \quad \{E\} \text{B}()() 
\]

Option 2: Use the UML 'interrupt' construct

\[
\text{A} \rightarrow \text{B} 
\]

The POOSL code in this case is of course identical to the code used in the first option.

Note: For extra clarity that execution resumes in state A when B (or consecutive states) finishes, one might consider adding the history state as the place where the interrupt handling finishes. The history state is drawn as a circle, with an 'H' in it.
4 The modelconvert tool

4.1 Model authoring

4.1.1 UML

There are all kinds of tools for drawing UML diagrams. Some are complete Computer Aided Software Engineering (CASE) tools, some are just drawing programs. For the UML to POOSL conversion tool, only drawing capabilities are needed in a tool. Appendix B describes the research of those tools. The main conclusions of this research are given here.

The authoring tool to use, must be cheap, because the model conversion is developed for use in a college environment, and students usually don't have much money to buy expensive software packages. To prevent customer lock-in, it was chosen to use a tool that supports the standards. In the future, more and more tools will be able to generate the standard UML export format, XMI[UML].

There are problems, but these exist in all tools and cannot be solved by taking another tool:

- Diagram information (the positions and sizes of elements) is not standardised yet in XMI. There is a specification on its way to standardise it.
- Parts of the UML to POOSL mapping (chapter 3) can only be done with UML 2, which is not totally supported by a lot of tools.

During development, the tool used was Poseidon for UML, version 2.1.1. Near the end of the period, a new version came out with support for drawing ports on objects and components, 2.1.2. The modelconvert tool does not support those last-minute additions. Future work includes reading in architecture information from the UML models and converting them to POOSL.

4.1.2 POOSL

Although the Rotalumis syntax was used in the previous section, the tool only supports SHESim-syntaxed POOSL in the POOSL to UML conversion, because SHESim is still the preferred application for authoring POOSL. But because it is easy to translate SHESim-POOSL to Rotalumis-POOSL using syntaxtranslator, this should impose no problems. See table 4 in paragraph 4.4.6 for a list of the syntactic differences.

SHESim is a graphical tool for authoring POOSL, and this makes it easy for users to quickly model a system. It also makes it easy to import parts of other models into a new model. The tool is based on this concept, as it only converts classes and not systems.

The POOSL tools have some important drawbacks together:
- SHESim is no longer developed and uses a strange (partly incorrect) syntax
- SHESim also includes diagram information (position and size of elements) in the model file
- Rotalumis has a clean syntax, but lacks any form of debugging or easy authoring methods

Future work should be aimed at creating a new version of Rotalumis, which makes it easy to draw end debug POOSL models, but should also be able to invoke Rotalumis and generate POOSL code files. An example addition of the language could be interface information, which must be checked against when authoring POOSL.

4.2 Usage

The modelconvert tool is a command-line application, which takes options to do what the user
The modelconvert tool wants. Having just the converter would get you nowhere; you will need UML and POOSL tools to import and export the models you created. In testing the tool the following model tools were used:

- **SHESim** for creating the POOSL models. SHESim has several ways of saving models, which can be very useful, because it makes it possible to only im- or export parts of the model. Important drawback of SHESim is that it is not being developed any more and has some changes in the syntax of the language compared to Rotalumis.

- **Poseidon** for creating the UML models. This was chosen because of the XMI[UML] support, although any tool that supports XMI1.2[UML1.4] should be good to use. See 4.2.1 on how to use Poseidon so that models can be converted to POOSL.

### 4.2.1 Creating the UML model

As already stated in 1.2 and 2.3, not all UML models can be translated to POOSL, but only a subset of it. This paragraph describes what must be in the models to convert them to POOSL. When it is known up front that a model will be converted later in the design process, the designer should make convertible models. If not, models have to be transformed first before conversion can take place. This transformation can mean that parts of the model will be unused in the conversion to POOSL.

Only class diagrams can be converted, other diagrams and model elements are ignored. In the class diagram the following issues need to be addressed:

- **Class prototyping** – Classes must be prototyped “process” or “data” for conversion. Classes that are not prototyped are ignored in the conversion.

- **Parameters** – Process methods do not have return values and the UML method consequently can also not have them. They should be modelled having the “in” or “out” kind.

- **Initial method call** – Process methods must have an initial method call; either inherited from a generalisation, or defined in itself. When no initial method call is found, a (non-fatal) error occurs in the conversion.

Because all POOSL models can be converted back to UML, there are no special issues in creating POOSL models.

### 4.2.2 Invocation

`modelconvert` is invoked in the following way:

```bash
java modelconvert <options> <model>
```

Where `options` is one of the following:

- `-v` Be verbose: give some explanation on what is being done
- `-u` Convert to UML (default: auto)
- `-p` Convert to POOSL (default: auto)
- `-x roundtripfile.xmi` Use `roundtripfile.xmi` as model/diagram repository
- `-o output` Use `output` as outputfile(s) (default:auto)
- `-d` Show debugging information
- `-t [target]` Define target (only applicable with -p switch):

---

9 UML parameters have a “kind”, which can be “in”, “out”, “return” or “inout”, describing the direction of the parameter.
The modelconvert tool

```
-v
  Be verbose: give some explanation on what is being done

shesim
  Create .psl and .sim files

shesimcl
  Create only .psl files

rotalumis
  Create .poosl rotalumis file (default)
```

And `<model>` can be either a UML or POOSL model:

- When converting from the POOSL that SHESim created, two files must be given. This is done by giving the name without the _Pr.psl and _Dt.psl suffixes,
- The POOSL parser does not support Rotalumis-style .poosl files, so it is not possible to feed those to modelconvert (the converter will exit with a POOSL syntax error),
- When converting from UML, only XMI1.2[UML1.4] is accepted. For example created by Poseidon for UML. Note that in this case the entire file name needs to be given,
- Ambiguous options for a specific model conversion are ignored and defaults are taken.

### 4.2.3 Typical use

There are different uses for modelconvert. You can for example start with a SHESim POOSL model and create a UML class diagram from it, let's say for documentation purposes. This paragraph however shows a typical design cycle for a system. Such a cycle starts with a UML model and uses several iterations over POOSL to check its correctness and finalise the design.

Figure 6 shows how the tool is used in a typical case, with in- and output files and where the tools are used. The cycles are described by the following steps:

1. Create a model in a UML tool as first system specification, with which you discuss about your system with fellow engineers. When such a model has evolved enough to be checked in POOSL, export it to an XMI[UML] file called `design.xmi`\(^{10}\)

2. Use modelconvert to convert the UML model into POOSL classes with the following command:

   ```
   java modelconvert -p -t shesimcl design.xmi
   ```

   This will create two files: `design_Pr.psl` and `design_Dt.psl`, containing the process classes and data classes, respectively.

3. Start a new SHESim session and import the two .psl files into the image (by clicking right-mouse-button on the class browsers). Use the imported classes to create your system: make class instances, channels, messages, edit (or create) source code, etc. When the system is to your liking, use the “Save all” option to save your system, give it the same name. This will create the following files: `design_Pr.psl`, `design_Dt.psl`, `design_CI.psl` and `design_TL.sim`.

4. Use modelconvert again to convert the POOSL back into UML. The original `design.xmi` is needed here for diagram information. Do this with the following command:

   ```
   java modelconvert -u -x design.xmi design
   ```

   Note that this overwrites your original XMI[UML] file, so make sure you always make a backup of it. You can also use the `-o` option to define a different output file.

\(^{10}\) This is an arbitrary name. Any file name will work.

Automatic version numbering can solve the overwriting of the XMI[UML] file.
5. Open the resulting XMI[UML] file into your UML editor of choice. All the changes made in the POOSL model should be in there now. Removed classes have the stereotype "removed" added, you need to remove them from the diagram manually. Edit further if needed.

6. Now you can convert the UML design back to POOSL again, the same way as in step 2. After converting, just overwrite the two .psl files. Make sure the _Cl.psl and _TL.sim are kept in place. They hold the cluster, system and diagram information, so you do not have to redraw all the channels and objects.

See appendix D for a case study, which is also used for testing the tool.

As said earlier, modelconvert can also be used to create UML models of already existing POOSL designs. To do so, just omit the -x option and modelconvert will create a clean XMI[UML] file, with no diagram information. You can just import that into your UML tool and start drawing the diagrams\(^\text{11}\). Elements such as generalisations and stereotypes will appear in the diagram automatically.

---

\(^\text{11}\) Poseidon has some weird behaviour in that you need to merge in an empty project first in order to be able to draw diagrams.
4 The modelconvert tool

4.3 Design

4.3.1 Overview

The modelconvert tool is developed for the conversion of UML models into POOSL models. It is a partial implementation of the mapping as described in chapter 3. It was not possible to implement the entire mapping because:

- Time was limited,
- Not all UML elements used in the mapping are available in the UML tool used.

Parts that are not in the converter have mappings that are only recommendations. The rest of the mappings is implemented and the choices made here are harder to change (or else a new implementation is needed).

The tool only converts classes and not their behaviour. The behaviour can be given in the code, either in SHESim, or in the UML tool you are using, but not using the dynamic diagrams given by UML. In itself this is not such a problem, as long as the code and the diagrams depict the same behaviour. However, it is very easy to corrupt that. A small change in code can mean a great difference in for example the state diagram. It is the responsibility of the engineer to keep the design consistent. See paragraph 4.5 for a discussion on how this can be solved in the future.

The modelconvert tool does not support packages (obviously, as they also do not have any meaning in POOSL; the internal UML model[12] though should support it) and classes that are defined inside packages will be ignored.

The tool is not considered ready for use so far.

4.3.2 Internal representation

The tool is written with extensibility in mind. It is written as generic as possible. This resulted in a well thought-out structure of the tool, that is easily extended. This is done by using internal models. Each modelling language can have its own internal model, with readers and writers for file in- and output. When conversion between modelling languages is needed, converter classes can translate between internal models. Because each internal model has its own reader and writer, the converters only need to interface with the internal models. See figure 7 for a schematic representation of this concept. When for example someone wants to implement a UML to SDL converter, he or she only needs to add an SDL internal representation and the converter. The entire internal UML model can be re-used.

The following paragraphs describe how the UML to POOSL conversions are made. They can be seen as an instantiation of the schematic given in figure 7. The actual implementation of the converter is not a UML to POOSL conversion, because UML has no defined file format. The format chosen is XMI[UML], hence the naming in the paragraph titles. The versions used are 1.2 for XMI and 1.4 for UML, as generated by Poseidon for UML 2.1.2.

[12] As described in 4.3.2.
4.3.3 XMI[UML] to POOSL conversion

Converting a UML model into POOSL can be done in only one way: as described in the mapping in chapter 3; it only accepts XMI[UML] files. The converter can output in several file formats: it supports SHESim .pss files and Rotalumis .poos files. The tool cannot convert complete models, but only class descriptions. SHESim has the option to load classes, Rotalumis needs a complete model to run. Translating a SHESim model to a runnable Rotalumis model can then be done with syntaxtranslator.

Figure 8 shows a schematic overview of the inner workings of the tool when converting XMI [UML] files to POOSL. Below is a short description of the several parts:

- **XML Parser/XMI[UML] Reader** The XML parser reads in the XML file and checks if the file is correctly formed. In this stage a DTD or XML Schema should be used too for validation, but those aren't available for the XMI[UML] version(s) used. The Document Object Model (DOM)\textsuperscript{13} tree is created in the XMI reader and from the DOM tree, the internal UML model is made.

- **UML to POOSL Converter** This is the actual converter that converts the internal UML model into an internal POOSL model.

- **POOSL Writer** This part writes the POOSL model to disk. It can either write SHESim or Rotalumis files.

Chapter 4.4 describes the parts in more detail, including used technologies and class descriptions.

\textsuperscript{13} More information on used technologies can be found in chapter 4.4.1.
4 The modelconvert tool

4.3.4 POOSL to XMI[UML] conversion

The conversion back to XMI[UML] is more complex. The main problem is how to keep the diagram information you created in the initial UML model. This is addressed by using the previously created XMI[UML] file as a diagram repository. See figure 9 for a schematic overview of the inner workings.

Not only the SHESim POOSL files are used in the conversion, but also an optional XMI[UML] file, from which only the diagram information is used. When that file is omitted, an empty XMI[UML] skeleton is created and used. This skeleton is in fact an empty UML model, in which later the classes are added. It consists of the XMI header and the UML:Model element. See appendix C for more detailed information on the structure of the XMI[UML] file format.

- **POOSL Parser** The POOSL parser reads in the POOSL code and transforms it into an internal POOSL model. The parser only supports class skeletons and parses the code...
The modelconvert tool

verbatim. No syntax checking or anything else is done on the code. Currently, the parser only supports SHESim syntaxed POOSL code, but can easily be changed to support other syntaxes.

- **POOSSL to UML Converter** The converter converts between internal models.
- **XMI[UML] Writer** This component creates a DOM tree from the internal UML model and the given XMI[UML] model or skeleton.

### 4.4 Implementation

For implementation, it is chosen to use the Java programming language. This language is chosen for the following reasons:

- **Platform independence** – the tool should run on all platforms, so that the designer of a system doesn't need to switch to a specific platform, just for converting his or her model.

- **Ease of implementation** – Java makes it easy to build large applications in a short matter of time. There is no need to bother about memory management or other time-consuming tasks.

- **Standards support** – Java has a fairly large base of tools supporting all kinds of standards, more on that in paragraph 4.4.1.

Of course, nothing comes for nothing:

- **Java is slow** – Converting a model can take some time. This is because it runs inside a virtual machine (VM).

- **Java is memory-hungry** – It can take several megabytes for converting a model. This is mainly due to the VM, but also to the internal modelling used. Models are read in entirely and take up a lot of memory.

Those problems are not large ones though. Converting models is not something is done very often and the time/memory taken by the converter are only fractions of what other tasks need.

#### 4.4.1 Used technologies and tools

Writing a tool that uses all kinds of standards and a fairly complex internal structure is not written with just a text editor and a byte code compiler. Several tools and technologies were used in the course of writing this piece of software. This paragraph lists all those tools and technologies and describes them shortly. You can skip this paragraph or read only parts of it if the given terms are already known.

**Document Object Model (DOM)**

The Document Object Model enables a user to access an XML file as a tree, which can then be traversed with for example XPath. When a DOM is made from an XML file, the user\(^\text{14}\) has an internal model of the XML file in memory.

More on the DOM can be found in \[A03]\.

**Document Type Definition (DTD)**

Because XML is very generic, a standard parser can only check the well-formedness of a document, not the validity. A document is well-formed when it adheres to the rules of XML. That is, tags are closed and nesting is done correctly. A document is said to be valid, when the

---

\(^{14}\) This can be either a programmer, or for example the user of a browser (Mozilla has DOM support built-in).
tags used are correct, as defined by another document. One way to describe this validity is using a DTD, which describes what tags can come where and what attributes may be used. Parsers that can also check against a DTD are called validating parsers. A DTD document is an instance of an SGML document.

More on DTDs can be found in [A03].

dom4j

The DOM implementation used for this project is dom4j. The main reason for that choice is the XPath support, which is not available in most other implementations. Another well-known implementation for example is JDOM.

More information and downloads can be found at [dom4j].

eclipse

Eclipse is a programming platform with integrated byte compiler, an editor with syntax highlighting and an execution environment. It was designed for Java, but can also be used for C++ development and other languages. To achieve that, it is equipped with a very good plug-in system. Eclipse is being developed by IBM and many open-source programmers worldwide.

More information and downloads can be found in [Eclipse].

eXtensible Markup Language (XML)

XML is a language to hierarchically describe data, derived from the more complex Standard Generalized Markup Language (SGML). It uses 'tags' to define the hierarchical structure of the data, which can be nested into other tags. The tags are written as normal text, enclosed in smaller-than «) and larger-than (») brackets. Inside the tags, it is possible to define attributes. Tags must be closed with a closing tag, which has the name of the opening tag, without attributes and with a forward slash (/) placed before the tag name.

A typical example of an XML file looks as follows:

```xml
<?xml version="1.0"?>
<car-collection>
  <car id="1">
    <brand>Volvo</brand>
    <type>C70</type>
  </car>
  <car id="2">
    <brand>Renault</brand>
    <type>Megane</type>
  </car>
</car-collection>
```

Because XML is a standard, all kinds of tools exist already, for example parsers. The parser used in the tool was Xerces (an open-source parser), but because the interface is a Java standard, any other parser will work just as well (or even better when faster for example).

An XMI[UML] file is an XML file, which is the reason the technology was used.

Java

Java is an object oriented programming language that was designed with platform
4 The modelconvert tool

independence in mind. The code, which resembles C++, is not compiled into machine native code, but to machine independent byte code that can be executed by a Java Virtual Machine (JVM). For all major platforms (Windows, Linux, Solaris, etc.) such a JVM exists and in this way, code written on Windows can also be executed on Linux.

As said, the language resembles C++ in syntax. Under the hood, however, there are some important differences. Java has no pointer support and you need not be worried about memory management. Instead, Java has a garbage collector that frees all unreferenced memory. Also, in Java, everything is an object. You cannot write procedural code with it.

Java has support for very many standards in the form of freely downloadable (often open sourced) libraries and classes. All in all, Java is a fairly easy language to learn and use, especially when one wants to learn object-oriented concepts. For more information on Java, see [Java].

**SHESim**

Creating the POOSL models was done in SHESim, or with a plain text editor, depending on the purpose of the model.

More information and downloads can be found at [POOSL]

**Poseidon for UML**

The used UML tool for this project is Poseidon for UML, mainly because its excellent standards support.

More information and downloads can be found at [Poseidon].

**XML Schema**

Another way of describing the validity of a document, besides DTD, is by the usage of XML Schema. An XML Schema document is in itself an XML document. XML Schema fixes some problems that exist in DTDs. It is, for example, possible to define the type of data that can occur inside a specific tag.

More on XML Schema can be found in [A03].

**XPath**

XPath is an XML standard, with which one can programmatically select one or more elements in an XML file. Selections can be made based on the name of the tag, the presence of a specific attribute or the position in the file. The selections (often references to it) can then be used to manipulate or read data. XPath is often used in conjunction with DOM, to traverse the tree, although it is not an object model.

More on XPath can be found in [A03].

See figure 10 for a schematic overview of all those used technologies and tools and how they relate to each other.
4.4.2 Tool class diagrams

Internal models

The design as described in paragraph 4.3 is implemented in Java with several packages and classes. Every internal model has its own package with all classes needed to describe such a model. Also, the file readers and writers for that model are described in there. See figure 11 for a complete class diagram of the internal models.

Most classes of internal model packages are specialisations of classes in the 'generic' package. That package describes generic object-oriented information, such as classes and methods. This is done so that generic parts are the same among all models. The generic classes are almost all defined as being abstract and thus cannot be instantiated.
Some of the UML classes have a separate stereotype attribute. It is not possible to define a UMLElement class and derive the other UML classes from it, because Java does not support multiple inheritance.

The readers of the internal models are the *Parser classes, which will create the instances of the model classes. The writers are implemented as a toString() method in the *Model classes. Those methods are default used by Java when concatenating an object to a string or for example printing it.
4 The modelconvert tool

Converters

The converters are placed in one package, 'modelconverter'. They consist of one converter class that is publicly available, and some converter-specific helper classes, only available to the package itself. They make use of the classes defined by the other packages to create the internal models. See figure 12 for the class diagram.

![Class diagram of the modelconverter package](image)

Figure 12: Class diagram of the modelconverter package

4.4.3 Importing models

UML

UML models are imported using the XML tools available to Java. Creating the internal model is done by the following code excerpt:

```java
XMIParser xmiparser = new XMIParser();
umlmodel = xmiparser.parse(xmifile);
```

Where `xmifile` is a Java `File` object, which points at the file containing the model. `umlmodel` is an object of the `UMLModel` class. The `parse()` method works as shown in figure 13.

Notes:
- check XMI - This does some rudimentary checks on the structure of the XMI tree:
  1. Check if the XML file has an XMI root tag,
  2. Check if that tag has a `xmi.version` attribute, with '1.2' as value
  When any of the two fails, an `InvalidXMIException` is raised.
- The creation of the UMLClass can raise a `UMLModelError` exception.

The creation of the UMLClass objects is shown in more detail in figure 14.
4 The modelconvert tool

**Figure 13: Sequence diagram of XMIParser:parse() method**

- A 'Document' object (in the dom4j package) is a reference to the DOM tree.
- The creation of the UMLModel object takes the DOM element as a parameter in the constructor.
- The 'Element' object that refers to the class that is being imported.
- After creation, it is added to the UMLClass object.

**Figure 14: UMLClass creation from DOM element**
The POOSL parser parses SHESim .psl files, and builds an internal POOSL model from it. Because SHESim saves process and data classes in different files, the internal POOSL model is built in parts. First the process classes in one model, then the data classes in another. Afterwards, both models are merged together:

```java
POOSLParse parser = new POOSLParse();

File f = new File(basename + "_Dt.psl");
if (f.exists() && f.canRead()) datamodel = parser.parse(f);
if (f.exists() && f.canRead()) processmodel = parser.parse(f);
pooslmodel = POOSLModel.mergeModels(processmodel, datamodel);
```

Where the datamodel, processmodel and pooslmodel objects are of the POOSLModel class. The parse() method reads in tokens, which can be words or symbols. Based on the read tokens, decisions are made. See figure 15 for the activity diagram of the class parse routine.

![Figure 15: Activity diagram of POOSL class import](image)
Whenever a syntax error occurs during parsing, the parser raises a POOSLParserException. The line and pos attributes of the parser then point at the exact position of the error.

The model merging is done in the static method `mergeModels()` in the `POOSLModel` class. It creates a new internal POOSL model and copies all classes, warnings and errors from both models at the input into that model.

### 4.4.4 Converting models

#### POOSL to UML conversion

The POOSL to UML conversion is done in the `POOSL2UMLConverter` class. Figure 16 shows how it is done in a sequence diagram. The call to `convertParents()` creates the references to parent classes in the UML model.

![Sequence diagram of POOSL2UMLConverter::poosl2uml() method](image)

**Figure 16: Sequence diagram of POOSL2UMLConverter::poosl2uml() method**

#### UML to POOSL conversion

The `UML2POOSLConverter` class converts objects of the `UMLModel` class into objects of the `POOSLModel` class:

```java
POOSL2UMLConverter converter = new POOSL2UMLConverter();
umlmodel = converter.poosl2uml(pooslmodel);
```
Inside the `poosl2uml()` method a new `POOSLModel` object is created and given the same name as the UML model. Figure 17 shows how the conversion is done in the `UML2POOSLConverter::uml2poosl()` method.

![Sequence diagram of UML2POOSLConverter::uml2poosl() method](image)

Figure 17: Sequence diagram of UML2POOSLConverter::uml2poosl() method
4.4.5 Exporting models

UML

The export is done with the `UMLModel::asXMI()` method. This method takes an optional object of the `File` class, denoting the round trip file and returns an object of the `Document` class of the dom4j package, which is a DOM tree. That DOM tree can then be written to an XML file with the `XMLWriter` class of the dom4j package. See figure 18 for the activity diagram of the `UMLModel::asXMI()` method.

![Activity diagram of UMLModel::asXMI method]

Figure 18: Activity diagram of `UMLModel::asXMI` method
First, two checks are done to ensure the XMI file integrity:

- Stereotypes needed for POOSL modelling,
- Data classes needed for POOSL modelling.

When those are not present in the round trip file, they are added. Second, the classes that were in the round trip file, but not in the UML model that is being written back, are stereotyped «removed», for later manual removal in the UML authoring tool. They can not be removed by the tool, because that can break diagram or model consistency.

Finally, classes are converted (when differing from what is in the round trip file) or added to the DOM tree. This is done as depicted in figure 19.

![Diagram of conversion of UML classes to the DOM tree](image)

*Figure 19: Conversion of UML classes to the DOM tree*

The `ConvertUMLClassElement` and `CreateUMLClassElement` can raise a `ClassNotFoundException`, when an undefined class is encountered during creation or conversion of the class. The `convert_classes` method catches that exception and handles the undefined class first, in a recursive way.

**POOSL**

The POOSL export supports two file formats: SHESim .psl files and Rotalumis .poosl files. The SHESim file format support is done by the `WriteSHESimPOOSL()` method and the Rotalumis file format support makes use of the overloaded `toString()` method. Both methods use the `classString()` method to export POOSL classes. This method takes a boolean parameter, denoting if the export is to be Rotalumis syntax (true) or SHESim syntax (false). The `classString()` method of the `POOSLDataClass` is given in figure 20. Figure 21 shows the `POOSLProcessClass::classString()` method.
The errors and warnings are annotated in the exported file, and consolidated in `errors` and `warnings` attributes of the `POOSLModel`.
4.4.6 Caveats

Classes not yet converted (tree colouring)

When converting classes, they are all considered in a linear way: the classes are in a linked list and are converted one-by-one. When converting classes this way, things go wrong when a class references another class that is further away in the list, for example in a specialisation-generalisation link. This is shown in figure 22. If that is the case, the referenced class needs to be converted first, before the reference can be made.

This problem can be solved by first converting the reference that is made (Class C), and afterwards return to the referring class (Class A) and continue the list. This, however, introduces new problems:

- What if the referenced class also references another class?
- What happens when the list is traversed further and the referenced class is converted again? Will it be overwritten, or defined twice?

Those problems are solved by recursion and tree-colouring. Recursion is used to solve the first problem. Before converting a class, its references are converted in a recursive way, so we always will convert the class that is “highest” in the hierarchy.

Tree-colouring is used to prevent double conversion. Whenever a class is converted successfully, it is coloured “black” (classes are “white” by default). Consequently, when a “black” class is encountered in traversing the list, it is safe to ignore it. To implement this, an extra attribute was added to each class, named “colour”\(^{15}\).

Syntactic differences between Rotalumis and SHESim

Because POOSL is in active development, there are syntactic differences between the code used by Rotalumis and SHESim. The POOSL syntax of Rotalumis is documented in [vB02] and is the latest version. The modelconvert tool can export to both Rotalumis and SHESim style POOSL, but only import SHESim style POOSL. The following table shows the differences between the two syntaxes.

\(^{15}\) In the code, this was made an Integer, with 0 being white, 1 being black. Other implementation are possible of course.
4 The modelconvert tool

Table 4: Syntactic differences between SHESim and Rotalumis

<table>
<thead>
<tr>
<th>SHESim</th>
<th>Rotalumis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variables divided by &quot;,&quot;.</td>
<td>Instance variables divided by &quot;,&quot;.</td>
</tr>
<tr>
<td>Superclass defined in the form</td>
<td>Uses extends keyword to define super classes.</td>
</tr>
<tr>
<td>/* superclass: (&lt;classname&gt;) */</td>
<td></td>
</tr>
<tr>
<td>just below the class name.</td>
<td></td>
</tr>
<tr>
<td>Order does matter in files,</td>
<td>Order doesn't matter concerning super classes.</td>
</tr>
<tr>
<td>concerning superclasses (classes</td>
<td></td>
</tr>
<tr>
<td>get overwritten while loading, this</td>
<td></td>
</tr>
<tr>
<td>resets any superclass identifier in</td>
<td></td>
</tr>
<tr>
<td>other classes).</td>
<td></td>
</tr>
<tr>
<td>Needs the following extra keywords:</td>
<td>Doesn't need any channel/message information</td>
</tr>
<tr>
<td>communication channels</td>
<td>in class definition, it is defined in the code</td>
</tr>
<tr>
<td>message interface</td>
<td>itself.</td>
</tr>
<tr>
<td>Expression lists may end with</td>
<td>Generates a syntax error when an expression</td>
</tr>
<tr>
<td>a semicolon.</td>
<td>list ends with a semicolon.</td>
</tr>
<tr>
<td>(e.g. abort S1 abort S2)</td>
<td>(e.g. abort S1 with S2)</td>
</tr>
</tbody>
</table>

FW Those problems immediately introduce important future work. POOSL needs to have one well-documented syntax. Because the Rotalumis syntax is clean and correct (there are faults in the SHESim syntax, for example the use of 'communication channels', where in fact ports are defined), the standard syntax should be based upon that. Such a standard will also lead to easier acceptance by the software/hardware engineering community, which is one of the current goals.

4.5 Future enhancements

FW The tool now only supports class diagram conversion. This is a good start, but not nearly ready, a lot of additions and enhancements must be made before the tool can really be useful in a production environment. This paragraph shows some of these additions and enhancements and discusses how or when they can be implemented.

Architecture modelling

An important part of a POOSL model is the top-level system specification. This is in fact an object diagram, because it consists of instances of process classes and channels. Such a specification can be modelled in several ways, for example as a object diagram with a specific name. Closely related to the top-level specification are the cluster class architecture diagrams. Here, the same issues apply. To correctly implement architecture modelling, more research is needed in how it should be modelled/mapped and how that is implemented in XMI[UML]. The mapping given in chapter 3 can be used in this research.

Behaviour modelling

Besides code skeleton generation, also conversion of behaviour should be added. Chapter 3.4 discusses some issues related to that. It needs more research before it can be implemented in the tool. Besides just the UML to POOSL mapping, research is needed on how it is

16 This is different from the POOSL syntax given in [vB02], where port and message interface are given explicit in the process class header (just before 'instance variables').
implemented in XMI[UML]. It is unknown to what extent POOSL to UML conversion is possible of behaviour. On first sight it seems very complicated to generate state diagrams from POOSL code, if not impossible. The main reason for that is that a state diagram can be modelled in code in several ways. Detecting what method was used and acting upon that can be extremely hard.

**Executable models**

When both architecture and behaviour modelling are implemented in the tool, it might be possible to integrate it into a UML program and create executable UML models in this way, without even having to write or even see POOSL code. Another option is that a tool emerges, with which POOSL models can be designed, based on a subset of UML. This would create correct UML models, that have the POOSL semantics and can be executed inside the program. Where the development will lead us, nobody knows, but there still is a lot to do.
5 Conclusions and future work

5.1 Conclusions

Creating POOSL models from UML is very well possible. The main reason is that both languages are based on the same concepts and earlier work. POOSL however took the way of executable models, whereas work in the UML field was aimed at completeness and definition of models, mainly for communication among engineers. The fact that both languages started growing apart meant that some concepts need extra attention before they can be converted. Think of the active process classes and the architecture model as good examples. The newest version of UML, 2.0, solves some, but not all, of these problems. Luckily UML has ways of extending the language, so that even the concepts that have no direct mapping can also be converted in a clean way.

Having a correct mapping between the two modelling languages creates extra possibilities for both sides. At the POOSL side, this can boost popularity of the language and with that, more developments of tools and standardisation. The popularity boost can put POOSL to the test, so that it can be seen if it can handle different kinds of usages. For UML, this mapping creates a way to add semantics to models and open a way for executable models, something the industry is starting to introduce. This is not limited to POOSL, because it is possible to create C++ code from POOSL, so now C++ can be generated from UML, all with open and free tools, which is an important development.

The modelconvert tool implements part of the mapping, which creates an actual start for all the things discussed above. It already has value for educational ends, being able to map classes from UML to POOSL. Besides that, it can create UML models from existing POOSL, enabling engineers to quickly generate documentation. By using standards throughout the development of the tool, it can more easily be extended or edited by other developers.

5.2 Future work

5.2.1 Mapping

The above conclusions already touch some ideas on how the research and the tool can progress in the future. The mapping currently is ready for static (structure) diagrams, on the M2 meta-level. First thing to work on is finishing the mapping on that meta-level, starting with the dynamic (behavioural) diagrams. Some ideas about that are already stated in this thesis. These can be used and need testing then, or all new ideas can be posed when these are not good enough.

Following this research (or concurrently with it) a mapping needs to be made on the M3 level, creating a real POOSL MOF meta-model and inserting all existing ideas into the OMG standards and profiles: Research the relation between the OMG action semantics of UML and POOSL.

5.2.2 Conversion tool

The conversion tool lacks two parts of the mapping: architecture and behaviour. The architecture is mapped already to UML 2, but no tools support it yet. In the future such tools will emerge and the tool needs adaptation to those new tools. The behavioural mapping is not finished yet, but when it is well thought-out, work can start with implementing that mapping into the tool. For that extra research is needed on how to read the XMI[UML] files. Besides this, work is to be done on the diagram information of the UML models: how XMI[UML] saves this information and how it can be manipulated to reflect the changes made in the POOSL model.
New features such as version numbering for file output must also be included.

### 5.2.3 POOSL

A very important thing POOSL needs for the future is a new standardised syntax. It should be based on the syntax Rotalumis uses, because of the reasons given in 2.2.2. The following concepts could be added for more consistent model design:

- **Interface definition** – Define interfaces for process classes, in the same way SHESim does that, but with the following changes:
  - Use correct keywords, not communication channels when in fact ports are meant.
  - Research existing interface definitions in other languages for ideas.
  - Enforce the usage of those interfaces. Give a syntax error when a message is sent over a port that is not defined in the interface definition.

- **Consistent syntax** – In the Rotalumis syntax, in the top level specification, ports and channels are placed differently from the cluster specification, while the top level specification and clusters are virtually the same, from an architecture point of view.

When POOSL is standardised, work can start on defining a complete MOF meta-model, including behaviour modelling and an XMI[POOSL] file format. The XMI[POOSL] file format should then be used as the default file format for POOSL models. This has serious impact on the conversion tool, because it can be changed into a generic XMI[POOSL] to XMI[UML] conversion, which can be done with existing tools.

### 5.2.4 POOSL tools

The POOSL tools (SHESim and Rotalumis) need serious attention, too. A whole new platform should be developed, which uses UML style diagram drawing, so that there is no need any more for conversion. This new platform could use Rotalumis for high-speed or real-time execution of the models and should have the following features:

- Syntax highlighting when editing POOSL code,
- Good error checking on the code (think of code consistency, as defined by the POOSL standard; see also above),
- Generation of clean POOSL code (and C++ if possible),
- Generation of XMI[POOSL] files,
- Generation of XMI[UML] files.

All this will take a lot of time to complete, and the developments done in the UML field need to be closely watched. By the time all this is finished, POOSL and UML will have been converged to a single real-time embedded modelling platform.
# A List of abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
<td>23</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
<td>30</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
<td>30</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended BNF</td>
<td>11</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td>32</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
<td>7</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
<td>7</td>
</tr>
<tr>
<td>POOSL</td>
<td>Parallel Object-Oriented Specification Language</td>
<td>8</td>
</tr>
<tr>
<td>SHE</td>
<td>Software/Hardware Engineering</td>
<td>2</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
<td>12</td>
</tr>
<tr>
<td>XMI</td>
<td>XML-based Metadata Interchange</td>
<td>8</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
<td>31</td>
</tr>
</tbody>
</table>
B UML authoring tools

To draw UML diagrams, several tools are available. This chapter describes some of those tools and looks at what is available (diagrams, exporting, price, etc). Goal of this small research is the selection of such a tool, for use in an educational environment. The POOSL generation will be based on the tool selected.

The assessed tools are: ArgoUML by Tigris, Poseidon for UML by Gentleware and Visio 2002 by Microsoft. Other proprietary products, such as iLogix Rhapsody and Rational Rose are not looked at, because of licensing issues\(^\text{17}\). Those issues don’t apply to Visio, because a license is available for EE students.

B.1 ArgoUML

ArgoUML is an open-source UML tool written in Java. A free download can be obtained from http://argouml.tigris.org. It is open sourced, so whenever bugs or shortcomings are found, they can be resolved. ArgoUML is just in version 0.14 which means it might not be ready for production tasks\(^\text{18}\).

B.2 Supported UML diagrams

Class, Use Case, State, Activity, Collaboration and Deployment diagrams are supported. Not being able to make sequence diagrams can be annoying, especially when needing to define timing, but UML mostly relies on class and state diagrams.

**Class Diagrams** – Class diagrams support everything which is needed in these diagrams. It is easy to add methods and attributes and all standard primitives are supported.

**Use Case Diagrams** – Use case diagrams are fully supported.

**State Diagrams** – State diagrams are bound to classes or use cases and are fully supported.

**Activity Diagrams** – Activity diagrams look like state diagrams and are also fully supported.

**Collaboration Diagrams** – Those aren’t very intuitive. Objects cannot be bound to classes and message directions cannot be defined. This is an important drawback, because collaboration diagrams can be used well in real time embedded systems, to create top level design.

**Deployment Diagrams** – Seems to be fully supported.

B.2.1 Standards compliance

ArgoUML is open sourced and those applications usually use open document formats. So does ArgoUML; it saves its model as a .zargo file, which is a ZIP compressed file, containing project information and an XMI 1.0 model definition for UML 1.3 (last is 1.5, 2.0 is in the works).

B.2.2 Features

- ArgoUML has an automatic ToDo list, which points you at things that still needs attention (for example element naming)
- Java code generation is integrated, though it seems limited to class template generation.
- Plugin support. Java code generation is default, C++, C# and PHP are offered separate. A POOSL code generation tool can be made as such a plugin (or extension, as they call it). This renders the need of XMI[UML] as not-needed\(^\text{19}\).

---

\(^\text{17}\) The product has to be used by students, who do not want to pay for their software. Or maybe just a little.

\(^\text{18}\) According to the website http://argouml.tigris.nl, it isn’t.

\(^\text{19}\) An XMI[UML] to POOSL code generation tool is better though, because it doesn’t rely on any tool.
B UML authoring tools

B.2.3 Caveats
ArgoUML has some important drawbacks, the most important being the sometimes awkward usage:

- Weird in ArgoUML is the way to add stereotypes: first, you have to give an element a stereotype (any will do). Second, you can choose 'New stereotype'. Stereotypes are stored in the model, so when specific stereotypes are needed, a template project can be made first, with all the needed stereotypes (and maybe classes) defined.
- ArgoUML will jump to the ToDo list at weird times. That makes the usability a bit messy (that is, you click a lot).
- Some things sometimes just stop working (in the entire program, for example not being able to begin a new model). Also, ways how to select or edit things are not intuitive.
- Just plain annoying is the fact the ArgoUML doesn't remember the working directory. It starts in My Documents all the time.
- No undo support.

Furthermore, there are some things that you can not do in ArgoUML, which are very useful when modelling real time systems:

- It is not possible to add constraints to your diagrams.
- Timing must be in a note, which introduces restricting syntax.

Then, there are those other things:

- Because it is in Java, it is not very fast and needs a lot of memory (a footprint of 40M when just started).
- It seems hard to open XMI/UML files created by other (or even ArgoUML itself!) UML tools. Sometimes it opens only the classes, sometimes the entire model. Editing seems not to work.
- Creating a collaboration diagram is not easy. Objects cannot be bound to a class.

B.2.4 Conclusions
In conclusion, ArgoUML is usable for use in a educational environment. The most important drawbacks being the absence of certain diagrams and the usability issues.

Creating a code generation tool will introduce problems for the top level design part, as collaboration diagrams aren't fully supported. However, if classes (both data and process) can be modelled entirely (that is including method bodies), making the top level specification is relatively easy with for example a tool like SHEsim, especially for small systems (which will be the default in an educational environment).

Pros

- Free as in beer, so ideal for students
- Free as in speech, so easily extended with missing features
- Platform independent

Cons

- Based on Java: slow and very memory consuming.
• Not all diagrams supported
• Serious usage problems

B.3 Poseidon for UML

Poseidon for UML is a commercial descendant of ArgoUML. It is released by Gentleware. It comes in different editions, of which the free Community Edition is looked at here. Other versions are Standard, Professional and Embedded Edition. It is written in Java and thus platform independent. The Standard Edition is more usable and can be obtained from Gentleware.

B.3.1 Supported diagrams

All diagrams are supported, but the deployment, object and component diagrams are tied together in one diagram.

Class diagram – All elements seem to be supported. Also constraints and stereotypes. Drawing a diagram is more intuitive and robust than in ArgoUML.

Use Case Diagrams – Use case diagrams are completely supported, including extension and inclusions. Activity and state diagrams can be connected to use case diagrams.

State Diagrams – State diagrams are supported completely, including history states.

Activity Diagrams – State diagrams are fully supported.

Collaboration Diagrams – Fully supported, the weird thing in ArgoUML that you cannot assign class types to an object is resolved here.

Deployment/Object/Component Diagrams – Although this is a three-in-one diagram, several separate diagrams can be created. All elements of the diagrams are supported.

Sequence diagrams – Sequence diagrams are fully supported. Drawing them can be a bit awkward sometimes, it is important to make the connections correctly. It is not possible to add information to sequences, for example timing constraints.

B.3.2 Standards compliance

Poseidon for UML has native XMI support. It writes XMI 1.2 for UML 1.4 and can read XMI 1.0, 1.1 and 1.2 (UML versions unknown). Diagrams though are not imported, just the classes and their relations. Poseidon’s own file format is a .zuml file. This is a ZIP compressed file with project information and an XMI[UML] model.

B.3.3 Features

Poseidon is derived from ArgoUML, but has some important improvements:
• Adding stereotypes is easier than in ArgoUML.
• The strange usability issues that exist in ArgoUML are not in this tool.

Drawing is intuitive:
• New elements are created when a link is extended.
• Browsing through properties is easy.
• Critiques that create a ToDo list (must be enabled explicitly).

20 See B.3.5 for more information.
• Poseidon for UML seems to implement UML correctly.
• Poseidon has native XMI support (XMI 1.2 for UML 1.4 according to the documentation). This means that when a code generation tool is made with these standards, it will be usable with future versions of other tools.
• Free as in beer, at least the community edition is.

B.3.4 Caveats

The Community Edition of Poseidon was tested, which lacks a few features which are in the full editions:

• No printing support, this is only available in the commercial editions. A work-around however is per-diagram export to PDF, and print from a PDF viewer. Other formats (GIF, JPEG, SVG among others) are supported too.
• Turning off single critiques.
• Browsing critiques by diagram.
• No Copy-Paste support.

Furthermore:

• Just as ArgoUML, it is written in Java (though released as a Windows executable\(^{21}\)), it consumes a lot of resources. It is not very speedy and has a memory footprint of 72MB (approaching 100MB after some use), just after startup.
• Sometimes, repainting is not done, which may be confusing because elements appear to be on the wrong places. Resizing the offending element repaints and makes the appearance correct again.

B.3.5 Licensing

Poseidon is not freeware. There is a free Community Edition, but this lacks some features. Gentleware however offers some licensing for students. The following table shows the possibilities:

Table 5: Poseidon editions

<table>
<thead>
<tr>
<th>Edition</th>
<th>License</th>
<th>Discount</th>
<th>Price (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Standard</td>
<td>0.00%</td>
<td>199</td>
</tr>
<tr>
<td>Professional</td>
<td>Standard</td>
<td>0.00%</td>
<td>699</td>
</tr>
<tr>
<td>Embedded</td>
<td>Standard</td>
<td>0.00%</td>
<td>1249</td>
</tr>
<tr>
<td>Professional</td>
<td>Students Special</td>
<td>0.00%</td>
<td>69</td>
</tr>
<tr>
<td>Standard</td>
<td>University, ≤ 12 licenses</td>
<td>50.00%</td>
<td>99</td>
</tr>
<tr>
<td>Standard</td>
<td>University, &gt; 12 licenses</td>
<td>80.00%</td>
<td>39</td>
</tr>
<tr>
<td>Standard</td>
<td>University, &gt; 24 licenses</td>
<td>90.00%</td>
<td>19</td>
</tr>
<tr>
<td>Professional</td>
<td>University, ≤ 12 licenses</td>
<td>50.00%</td>
<td>349</td>
</tr>
<tr>
<td>Professional</td>
<td>University, &gt; 12 licenses</td>
<td>80.00%</td>
<td>139</td>
</tr>
</tbody>
</table>

---

21 Poseidon is distributed as a collection of jar files, but no Java RE is needed. It looks like the Java VM is in the Poseidon executable.
When a university license for the standard edition is obtained, and more than 24 licenses are bought, only €19 is to be paid for Poseidon Standard, which is a very usable version. If a student wants a more sophisticated version, the student special is available for €69.

The differences though, between the Standard and the Professional Edition are mainly targeted at code generation. The Professional Edition has C# and VB.NET generation and MDL and JAR import capabilities.

### B.3.6 Conclusion

Poseidon for UML fixes a lot of ArgoUML’s problems and is very usable in a production environment, when the hardware allows you to run it. It seems like an ideal tool for educational environments, provided the print support isn’t needed very hard. When comparing the Community Edition to ArgoUML, Poseidon for UML turns out to be a lot better, both in usability and in diagram support.

**Cons:**
- A bit slow, large memory footprint
- No print support in free community edition

**Pros:**
- All diagrams supported
- Easy drawing
- XMI 1.2 for UML 1.4
- Free as in beer (community edition)
- Platform independent

### B.4 Visio 2002 Professional

Microsoft Visio is part of the Microsoft Office suite. It is a diagram drawing tool for all kinds of diagrams. UML modeling is provided as a part.

#### B.4.1 Supported diagrams

All except object diagrams, but object diagrams can be modelled using deployment diagrams. Visio has no diagram centred view. All elements can be drawn on one page, so it is important the designer makes clear distinction between the diagrams. This is very well possible by using the ‘model explorer’, at the lower left hand side of the program’s window.

Visio uses ‘stencils’ that group the different diagram elements together. Those stencils are not connected to a specific page.

**Class diagram** – The class diagram is fully supported. There are some things worth noting:
- The class diagram’s elements are stored in Visio’s ‘static structure’ stencil. And the diagram name in the model explorer is also ‘static structure diagram’.
- Constraints can be connected to any other element.
B UML authoring tools

- Built in support for some known constraint types (\{or\} and 2-element constraints).

Use case diagrams – Fully supported.

State Diagrams – State diagrams can be connected to classes. All state diagram elements are supported, including history states.

Activity Diagrams – Fully supported.

Collaboration Diagrams – Fully supported.

Deployment/Object diagrams – Fully supported.

Component Diagrams – Fully supported.

Sequence diagrams – Very freestyle drawing possible, which might introduce model errors very easily. Missing 'creation' and 'destroy' messages.

B.4.2 Standards compliance

XML[\text{UML}] support is not native, but an XML[\text{UML}] export library (.DLL) can be downloaded without support from the Microsoft website. A macro can then be built that uses functionality from that DLL. Source code for that macro is provided, but Visual Studio .NET is needed to compile. Some searching on the Internet though gave a pre-compiled macro, which was easily installed in Visio.

The exporter exports to XML version 1.0 for UML 1.1. However, the generated file is not even a valid XML file. When the XML errors are resolved, it doesn't seem to be a valid XML 1.0 file: when trying to open it using Poseidon, Poseidon hangs.

Visio can save diagrams as XML files, but the DTD used is probably proprietary and thus unusable; future versions might (probably will) break the specification and reverse engineering the DTD, for creating a code generation tool, is too much (useless) work.

B.4.3 Features

- All diagrams are supported and really all possible drawings can be made.
- Reverse engineering from MS Visual Studio (strangely enough, no forward engineering).

B.4.4 Caveats

- Drawing is not 'made for UML', but more generic for diagrams. Connecting and routing isn't as easily done as with Poseidon or Argo.

B.4.5 Conclusions

Visio 2002's UML capabilities seem to be for documentation only, and not code generation.

Cons
- Being able to draw everything can introduce model errors very easily.
- Old XML[\text{UML}] version
- XML[\text{UML}] export doesn't generate correct XML, or even correct XML.

Pros
- Everything UML is supported
- Usage probably well-known, because the tight integration with the rest of MS Office.
B.5 Conclusions

The three tools checked are all quite usable for creating UML diagrams. However, some more is needed. Some usability is always nice and the target is to create a UML to POOSL code generation tool and thus processing the model is an important factor. Using XMI[UML] to do so is a good solution for that problem. This means the tool has to support XMI[UML] export. Round trip engineering is also a goal of the project, which means XMI[UML] import is needed too. The following table shows a comparison between the three tools.

Table 6: Comparison of the tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Supported diagrams</th>
<th>Price</th>
<th>XMI support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>Cl, UC, St, Ac, Co, De</td>
<td>Free</td>
<td>native, version 1.0 for UML 1.3 read and write</td>
</tr>
<tr>
<td>Poseidon for UML</td>
<td>Cl, UC, St, Ac, Co, De/Ob/Cm, Sq</td>
<td>Several editions, the community edition is free. Cheapest standard edition is €19, cheapest professional edition is €69 (see B.3.5).</td>
<td>native, version 1.2 write for UML 2.0, 1.0 and 1.1 read</td>
</tr>
<tr>
<td>Visio 2002</td>
<td>Cl, UC, St, Ac, Co, De/Ob, Cm, Sq</td>
<td>Campus license already available for EE students.</td>
<td>macro, generates incorrect XMI in an incorrect XML file (XMI 1.0 for UML 1.1), write-only</td>
</tr>
</tbody>
</table>

Diagrams: Cl = Class, UC = Use Case, St = State, Ac = Activity, Co = Collaboration, De = Deployment, Ob = Object, Sq = Sequence, Cm = Component.

According to this table, Poseidon seems to be the best choice. Especially the XMI[UML] support is important for that choice. Poseidon looks ready for future versions of XMI and UML. Only drawback might be the price here.
C XMI[UML] outline

XMI[UML] files are XML files, and this means it is possible to show the structure in the form of a tree. This appendix shows the structure of XMI[UML] files, as generated by Poseidon. XMI [UML] files are divided into three parts: a header, which has information on version and generator, a model, which has all the model information and a diagram part, that has all the diagram information of the UML model. This last part isn't standardised yet in the XMI[UML] file format. The future standard will most probably look like the structure that Poseidon uses.

As all XML files, also XMI[UML] file start with:

```xml
<?xml version='1.0' encoding='UTF-8'?>
```

Then, the root element of XMI[UML] files:

```xml
<XMI xmi.version='1.2'
    xmlns:UML='org.omg.xmi.namespace.UML'
    timestamp='Tue Oct 07 13:11:57 CEST 2003'>
```

As shown, the root element contains the XMI version used, as well as a namespace 23 definition, which is set to UML.

C.1 Header

Because the header is fairly small, it is shown here completely and is quite self-explanatory:

```xml
<XMI.header>
    <XMI.documentation>
        <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
        <XMI.exporterVersion>1.0</XMI.exporterVersion>
    </XMI.documentation>
</XMI.header>
```

C.2 Content

The next element in the XMI[UML] tree is the <XMI.content> element, which contains the entire semantic model. All unique elements have an xmi.id XML attribute, which is a unique identifier. This identifier is used in references to that element. Below is a tree of the XMI[UML] elements as used in the modeleconvert tool. There are some more elements, but they aren't considered in the conversion. Refer to the XMI[UML] DTD or look into an example file for more insight in that.

---

23 A 'namespace' is an XML concept, it describes a space in which certain elements are used. See [A03] for more information.
Figure 23: XMI[UML] file structure for the XMI.content element

Some elements have an identifier after the element name:

- 'name', is an attribute and refers to the name of that model element,
- 'idref', is an attribute and designates a reference to a unique element, somewhere else in the
model,

- "#text", refers to text inside the element tags.

### C.3 Diagram information

The last element inside the XMI tag are the UML:Diagram elements. There, the diagram information is defined, but note that it is no official standard yet, though it will be and the files that are generated by Poseidon are close to that future standard.

Every diagram has its own UML:Diagram element, which consists of elements that describe the position and size of the contained elements in that diagram. Those contained elements can then again contain other elements, etc. When an element refers to part of the semantic model, it is notated with the UML:GraphElement.semanticModel tag. Figure 24 shows all this in a schematic overview.

```
XML.content
  └ UML:Diagram name
      └ UML:GraphElement.position
          └ XML.field #text
          └ XML.field #text
      └ UML:GraphNode.size
          └ XML.field #text
          └ XML.field #text
      └ UML:Diagram.viewport
          └ XML.field #text
          └ XML.field #text
      └ UML:GraphElement.semanticModel
          └ UML:SimpleSemanticModelElement
      └ UML:GraphElement.contained
          └ UML:GraphNode
```

*Figure 24: XMI diagram information structure (part)*

Then, inside the UML:GraphNode element, all other elements (except diagrams) can be placed. In this way all diagrams can be defined. The XML.field elements have numbers in them, being X and Y coordinates.
D Case study

This appendix describes a case study, which is used to test the tool. It consists of the 6 steps, as given in paragraph 4.2.3. The case is based on a case which was done more extensively during the course of the research. The difference between that one and the one that is discussed here is the goal. Here the goal is showing how the tool works and how it is used in a typical case. The other case was to gain extra insight in the problems an engineer stumbles upon. That knowledge was used throughout the design and creation of the tool.

The case is about a two train tracks, which cross. There is need for a control system that makes sure the trains that ride the tracks never collide. After a brainstorm phase it is decided to fix this using a semaphore-like solution: whenever a train approaches the crossing, it asks permission to use it. If it is free, the crossing grants the train its usage. If another train is using the crossing already, the train will be denied access and the train will wait until the crossing is released by the train.

This case study assumes the usage of Poseidon as the UML authoring tool.

D.1 UML model

The first step to make is creating the UML model. There are two main players in this system: the crossing and the trains. The trains will be communicating with the crossing. A possible class diagram can be like the one given in figure 25.

![Initial UML model](image)

**Figure 25: Initial UML model**

A first glance notices a few things:

- Both classes are stereotyped as 'process', which will cause the converter to make them process classes in POOSL.
- There is no intelligence whatsoever. The train can only start and stop and the crossing can't even do anything.
We will use this fairly simple model to export it to SHESim and edit it further from there.

**D.2 UML to POOSL conversion**

First we need to save the model as XMI[UML] file. Do this by selecting *File → Export Project to XMI* in Poseidon; we will use *crossing.xmi* as file name here. Then, issue the following command to convert it into POOSL:

```
bart@gandalf:-$ java modelconvert -p -t shesimcl crossing.xmi

Model converter v0.2, Copyright 2004 Bart Friederichs
Converting crossing.xmi to POOSL
*** POOSL model had 1 warning(s) and 2 error(s).
*** They are annotated in the POOSL code, use your favourite editor
*** and a POOSL manual to fix them.
*** SHESim cannot import it.
```

Apparently there was something missing or wrong in the UML model. When investigating the *crossing_pr.psl*, it can be seen that there are no initial method calls set, which are obligatory for process classes. And, as the tool correctly tells us, SHESim cannot import it. Fixing this, though, is easy. Just add an initial method call to the classes in UML. This is done by stereotyping methods as 'initial'. When that is done, *modelconvert* will correctly convert the model and SHESim will be able to import the POOSL files.

**D.3 Importing POOSL into SHESim and editing it**

Now we can import the two .psl files into SHESim. Do so by right-clicking the process class browser and select 'load'. Select *crossing_pr.psl* and click OK. Do the same for the data classes. All the classes show up in the class browser and can be used in your model, see figure 26.

Note that when no code is supplied in the UML tool, *modelconvert* will insert a *skip* statement, so that SHESim can import it.

Now we can start adding code, messages, etc., which we will do now. I added the following to the model to make it work:

- A new 'SmartTrain' class, derived from 'Train', with all intelligence in it
- An instance variable 'state' in the crossing class, defining what state the crossing is in (whether or not it is available, or who has asked permission).
- Two channels to connect the objects together.
- The object communicate by sending the following messages: GRANT, DENY, REQUEST and RELEASE.
D.4 Converting back to UML

Now the POOSL model can be translated back to UML with the modelconvert tool. To do so, first save the POOSL design as separate .psl files by selecting File → Save all in SHESim, again, give it the name crossing. SHESim will then create four files: crossing_Dt.psl, crossing_Pr.psl, crossing_Cl.psl and crossing_TL.sim. We will run modelconvert in the following way (the original XMI[UML] file is re-used here and overwritten, make sure to make a backup first):

```
bart@gandalf:$ java model convert -u -x crossing.xml crossing
Model converter v0.2, Copyright 2004 Bart Friederichs
Converting crossing to XMI, using crossing.xml as roundtrip file.
```

crossing.xml is now a the converted POOSL model. It can be imported into Poseidon to do extra editing.

D.5 Importing the XMI[UML] and editing it

The now created XMI[UML] file can be opened in Poseidon by choosing File → Open project... Select XML Metadata Interchange (*.xmi) as file type and select crossing.xml. This will open the project, you will see the same diagrams as you already created in step 1. The model itself is updated correctly, only the diagrams aren't. That can be seen in the model browser in the upper-left corner, as can be seen in figure 27.

A few notes need to be made here:

- When methods are created in Poseidon, it will add the return value (void) automatically.
This is wrong however for POOSL process classes, as their methods have no return value. After the roundtrip, those return values appear boxed. Redrawing the class is the fastest and best way to fix that.

- When adding classes that have gotten generalisations, the generalisations will appear automatically.
- New classes and new methods and attributes in existing classes are not added to the diagrams. Again, (re)drawing the classes fixes that.

See figure 27 also for the new UML model.

![Figure 27: Class diagram of converted POOSL model](image)

Now, we can edit our model a bit further. Suppose for example that a fellow engineer wants to use packets, which carry a message, instead of the messages as defined now. To do so, we add a Packet data class, which has a Message attribute and edit the code to use this new class. See figure 28 to see how the class diagram looks like now.

The code changes in the existing process classes are as follows:

- The initial method calls set up the Package
- Sending a message is changed from channel!MESSAGE into p setMessage ("MESSAGE"); channel!m(p).
- Receiving a message is changed from channel?MESSAGE into channel?m(p|p getMessage() = "MESSAGE").

---

24 Warning: Do not use delete to remove the class from the diagram, this will also remove it from the model. Use Ctrl-X (cut) instead, that will only remove it from the current diagram.
D.6 Converting edited model into POOSL again

Save the model as XMI[UML] and convert it to POOSL, using the following command:

```
bart@gandalf:-$ java modelconvert -p -t shesimcl crossing.xmi
```

Model converter v0.2, Copyright 2004 Bart Friederichs

Converting crossing.xmi to POOSL

*** POOSL model had 0 warning(s) and 0 error(s).

Note that this will overwrite the existing crossing_Dt.psl and crossing_Pr.psl files. You should have made backups of these (or trust the tool blindly, but history has shown that that is not a smart thing to do).

Now, just open the crossing in SHESim and the new model (including the Packet data class) is available. If you edited the code correctly, it should load and run without any interference of the user. Note that the modelconvert tool doesn't detect any errors in the POOSL code, only in the model. When there are errors in the code, such that SHESim cannot import it, it will not import the model at all. It is recommended to write all code in SHESim and not in Poseidon, because of this reason. It is done here however to show that it is possible.

This is the last step in the cycle. Repeat steps 3 to 6 until the model is correct enough for implementation.

D.7 Final notes

This case study was a well-thought out example, which doesn't discuss problems you might encounter when using the tool. This last paragraph shows some issues that can happen when
converting models.

- The tool only takes the four defined primitives *Char*, *Integer*, *Boolean* and *Real* of POOSL into account. Any other primitives that are defined by SHESim must be explicitly exported into UML.

- When converting back from POOSL to UML, the diagrams aren't updated at all. The underlying model information though, is correct. The changes can be checked into the diagrams by redrawing the diagram elements by hand.

- Syntax errors in the code aren't encountered until classes are read in by SHESim. The tool has no syntax error checking for the code.
References

References


[vB02] Bokhoven, L.J. van, Constructive Tool Design for Formal Languages: From Semantics to Executing Models, Diss (PhD) Eindhoven University of Technology, Eindhoven (The Netherlands), 2002


[vG04] Gerwen, E. van, A Design Method for Predictable Real-Time Control, Diss (MSc) Eindhoven University of Technology, Eindhoven (The Netherlands), 2004

