Integrating model-to-model and model-to-text transformations in a model driven framework for UWE

Mathias Colpaert
Acknowledgement

I would like to thank a number of people who made this possible. First of all, thanks to my thesis promoters Heiko Desruelle and Frank Gielen. They helped pushing me in the right direction the whole year. Special thanks to Heiko for the almost weekly meetings and detailed follow up of my progress. I would like to thank Stéphane Bégaudeau for helping me solve some technical issues with Acceleo. After hitting a brick wall, his personal involvement helped unblock the situation. Also thanks to Gustav Janssens for checking the spelling of this thesis with eagle eyes! Last but not least, I would like to thank my parents. They gave me the opportunity to study engineering and always supported me in every way. Without them, none of this would have been possible.

Mathias Colpaert, May 2013
Copyright notice

“The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation.”

Mathias Colpaert, May 2013
Integrating model-to-model and
model-to-text transformations in a
model driven framework for UWE

by

Mathias COLPAERT

Master’s thesis submitted to obtain the academic degree of
master of computer science engineering

Academic year 2012–2013
Promotor: Prof. Dr. Ir. F. GIELEN
Advisor: Ir. H. DESRUELLE

Faculty of Engineering and Architecture
Ghent University
Department of Information Technology
Chairman: Prof. Dr. Ir. D. DE ZUTTER

Summary

Due to the large number of devices and programming languages, implementing an application is an expensive task. This thesis discusses the design and implementation of a model driven framework for web engineering. Several technologies and standards are combined in a proof-of-concept framework for UML-based Web Engineering. By using this framework, applications only need to be designed once for all platforms, which reduces design time and development costs.

Keywords

model, metamodel, M2M, M2T, UML, UML profile, ATL, Acceleo
Combining model-to-model and model-to-text transformations in a model driven framework for UWE

Mathias Colpaert

Supervisor(s): prof. dr. ir. Frank Gielen, ir. Heiko Desruelle

Abstract— Due to the large number of devices and programming languages, implementing an application is an expensive task. This article discusses the design and implementation of a model driven framework for web engineering. Several technologies and standards are combined in a proof-of-concept framework for UML-based Web Engineering. By using this framework, applications only need to be designed once for all platforms, which reduces design time and development costs.

Keywords— model, metamodel, M2M, M2T, UML, UML profile, ATL, Acceleo

I. INTRODUCTION

Over the last decades many web engineering methods have been developed.[4] These methods describe ways of modeling and implementing web applications, taking into account the specific concerns and challenges for web applications. Web engineering is combined with model driven engineering, in which models represent an application, rather than code. In this way, developers only need to design their application once and consequently the design cost is reduced. This paper focuses on the current tools and technologies for model driven engineering. We use them to build a framework to extend the UWE web engineering method. We take the first steps towards an UWE dedicated framework.

The rest of this article is structured as follows. First we give a short overview of related work about UWE and software engineering standards. Next we discuss how a proof-of-concept framework was designed and implemented. Conclusions are made based on this framework and we end with a vision for future work.

II. RELATED WORK

A lot of OMG’s (Object Management Group[6]) standards are used in this work. Model Driven Architecture is OMG’s interpretation of model driven engineering.[5] Computationally Independent Models (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM) are built and transformed. Each engineering method can define a number of CIMs, PIMs, PSMs and transformations between these model types. UML-based Web Engineering (UWE) is a web engineering method that has done so, defining a content model, a navigation model, a process model and a presentation model. The model syntax is based on UML (Unified Modeling Language) and is extended with a UML profile. An overview of the UWE models is given in Figure 1. First a content model is built to model the data of the application. A navigation model is derived from the content model. It represents all the nodes in the application and the navigation paths. The process model is used to model the logic of the application whereas the presentation model is used to model the user interfaces. An extended introduction to UWE is given in [2].

III. DESIGN AND IMPLEMENTATION

The goal is to create a model driven framework for UWE. The system can be used by developers to create and transform UML-models. It allows to define model types and transformations. These transformations must be executed independently of the framework. Creating a proof-of-concept is divided in three parts: creating a framework, creating a model-to-text transformation and creating a model-to-model transformation.

First, we created a PIM to PSM transformation for the UWE presentation model. In order to do so, a Platform Specific Metamodel was defined. The metamodel is aimed at HTML/CSS/JavaScript implementation language. It combines aspects which are typical for the programming languages with modeling concepts from the UWE platform independent presentation model. By doing so, no semantic information is lost in the transformation. Figure 2 shows a part of the PSM metamodel. It contains elements aimed at the target language (html, head, body, ...) and rich semantic concepts such as formHandler, to load, validate and submit content of a form. The first transformation converts the UWE model to a platform specific model of this type. The transformation is implemented in the ATL transformation language. ATL is part of OMG’s Eclipse Modeling Framework.[1] It allows to define model-to-model transformations, and to create a JAR which is able to execute the transformation independently of Eclipse.
Second, the PSM is transformed to code. The transformation creates a number of HTML and JavaScript files out of the model. The focus is on two semantic structures, menus and forms. The concepts modeled in the PSM can be implemented in several ways. This is a parameter of the transformation. The transformation is implemented by using Obeo’s Acceleo, a template based code generator. The structure of the Acceleo transformation is given in an activity diagram in Figure 3. The main module has a PSM as input, it creates a HTML file and a number of JavaScript files. Furthermore, the other elements of the PSM are passed to the component module, which generates more code. Some elements are passed to other modules (menu module, selection module). In a complete transformation, it might be more appropriate to create a module for each PSM metamodel element. However, in this case it is unnecessary. Acceleo allows the creation of an independent JAR that can execute the transformation.

Third, a framework to integrate both transformations is created. The framework allows to import UML models, metamodels and transformations. It is able to work with external transformation JARs in a general way. It is lightweight, when models or metamodels are imported, a reference to the model location is saved, but the model is not checked. The framework has a simple Model-View-Control structure and is implemented in Java.

IV. EVALUATION

A small application is built and transformed with the framework to test it. First, a project is created and the two transformations that where created before, are imported. Furthermore an application is modeled with UWE, a creating content, navigation and presentation model. The model is imported in the framework and transformed to a PSM, which is imported in the framework and transformed to code with the Acceleo transformation. Figure 4 shows one of the generated pages of the application. The way the menu is rendered, is a parameter for the transformation. Apart from the user interface, a JavaScript file is generated, containing code to handle the validation of the forms of the application.

We have evaluated how the execution time of the two transformations scales with the input model size. Both ATL and Acceleo do not seem to have any scaling problems. Furthermore the RAM usage was evaluated. We can conclude that for larger models the RAM usage does not increase notably.

V. CONCLUSIONS AND FUTURE WORK

After working with all these technologies and creating a proof-of-concept, a number of conclusions can be made. The Eclipse Modeling Framework is a strong basis for model driven engineering. It provides the tools to define metamodels, UML profiles and transformation between models or to code. However, there is still a lack of compatibility of the UML models with other UML modeling tools.

There is a lesson learned. Using UWE to model user interfaces does not seem to be the most appropriate way. The diagrams tend to get overloaded and are too abstract. The reason for this is that UWE is a broad modeling language, applicable for a range of devices such as Tablets, TVs, ... To model smartphone application user interfaces, a What-You-See-Is-What-You-Get method could be more efficient. In a future work, UWE could be narrowed down to a subset for smartphone applications. As a basis, the navigation model can be used. The model editor can be taken into the cloud, in a collaborative online modeling tool. Moreover, the transformations could be executed on a server instead of local.

REFERENCES

Combining model-to-model and model-to-text transformations in a model driven framework for UWE
Mathias Colpaert
Supervisor(s): prof. dr. ir. Frank Gielen, ir. Heiko Desruelle


Keywords—model, metamodel, M2M, M2T, UML, UML profile, ATL, Acceleo

I. INTRODUCTIE

De afgelopen decennia zijn er verschillende web engineering methodes ontwikkeld.[4] Een method is een welbepaalde manier om applicaties te modelleren en implementeren. Hierbij wordt er rekening gehouden met de specifieke noden en moeilijkheden die bovenkomen tijdens het ontwikkelen van webapplicaties. Web engineering wordt gecombineerd met model driven engineering, waarbij men een model maakt dat de applicatie voorstelt in plaats van code. Het model is platform onafhankelijk waardoor de ontwikkelaar de applicaties slechts eenmalig moet modelleren voor alle toestellen. Dit leidt tot lagere kosten tijdens het ontwikkelen. In deze thesis kijken we welke model driven technologieën we kunnen gebruiken om een framework te maken voor de web engineering methode UWE. We zetten de eerste stappen naar een framework specifiek voor UWE.

De rest van dit artikel is als volgt gestructureerd. Eerst geven we een kort overzicht van gerelateerd werk over UWE en software engineering standaarden. Vervolgens worden het ontwerp en de implementatie van het proof-of-concept framework besproken. We trekken onze conclusies en eindigen met een visie voor de toekomst.

II. GEBRUIKTE TECHNOLOGIEËN EN STANDAARDEN


III. ONTWERP EN IMPLEMENTATIE

We willen een model georiënteerd framework bouwen voor UWE. Het systeem zal gebruikt worden door ontwikkelaars van webapplicaties om UML-modellen te creëren en transformeren. Met het systeem moet men modeltypos (bvb. UWE content model, ...) kunnen definiëren met de bijhorende transformaties. Er kunnen ook transformaties zijn die code genereren. Alle transformaties moeten onafhankelijk van het framework uitgevoerd worden. Het maken van een proof-of-concept is opgedeeld in drie delen: creëren van het framework, creëren van een model-to-model transformatie en het creëren van een model-to-text transformatie die code geneereert.

Eerst is er een PIM naar PSM transformatie gecreëerd voor het UWE presentatie model. Om dit te doen, is er een Platform Specific Metamodel gedefinieerd waaraan het PSM moet voldoen. Het metamodel is gericht op HTML/CSS/JavaScript als implementatietaal. Het combineert aspecten van de implementatietaal met semantische modelleringconcepts. Op deze manier gaat er geen semantische informatie verloren bij de transformatie. Figuur 2 toont een deel van het gedefinieerde metamodel. Het bevat programmeertaalafhankelijke elementen.
In een tweede deel, wordt de PSM getransformeerd naar code. De transformatie creëert een aantal HTML en JavaScript bestanden uit het model. De focus van de tweede transformatie ligt op twee semantische structuren, menu’s en forms. De gemodelleerde concepten kunnen op verschillende manieren geïmplementeerd worden in de code. De manier waarop dit gebeurt is een parameter van de transformatie. De transformatie is geïmplementeerd m.b.v. Obeo’s Accelo. Accelo is een code generator gebaseerd op templates. De structuur van de implementatie is gegeven in Figuur 3. De main module heeft een PSM als input. Het creëert een aantal HTML en JavaScript bestanden. Vervolgens worden alle PSM component elementen doorgegeven aan de component module, die meer code genereert. Sommige elementen worden van de component module doorgegeven naar andere modules (menu module, selection module). De Accelo transformatie kan geëxporteerd worden in een onafhankelijke JAR.

In het derde deel worden beide transformaties geïntegreerd in een framework. In dit framework kan men UML modellen, metamodelen en transformaties importeren. Bij het uitvoeren van een transformatie worden de onafhankelijke JARs opgeroepen. Het framework is lightweight, modellen en metamodelen worden niet gevalideerd wanneer ze geïmporteerd worden. Er wordt een referentie naar de locatie van het (meta)model opgeslagen. Het framework heeft een simpele Model-View-Control structuur en is geïmplementeerd in Java.

IV. EVALUATION

Om te evalueren is er een kleine applicatie gebouwd en getransformeerd met het framework. Er wordt een project gecreëerd in het framework en de twee transformaties worden geïmporteerd. De applicatie wordt gemodelleerd met UWE en het content, navigatie en presentatie model worden gecreëerd. Dit model wordt geïmporteerd in het framework en getransformeerd met de eerste transformatie. Hierbij wordt een PSM aangemaakt. Deze wordt ook geïmporteerd en getransformeerd naar code met de Accelo transformatie. Figuur 4 toont een deel van de gegenereerde user interfaces. De manier waarop het menu weergeven wordt, is een parameter van de transformatie. Voorts worden er nog JavaScript bestanden gegenereerd, die logica bevat om de gemodelleerde forms te valideren.

V. CONCLUSIONS AND FUTURE WORK

Na het creëren van het framework kunnen we enkele conclusies trekken. Het Eclipse Modeling Framework is een sterke basis voor model driven engineering. Het voorziet de nodige tools voor het definiëren van (meta)modellen, UML profiles, model-to-model transformaties (ATL) en model-to-text transformaties (Accelo). Er is echter nog nodig aan een vlottere uitwisseling van UML modellen met andere modelleringsstools.

Verder besluiten we dat UWE niet de meest geschikte manier lijkt om user interfaces te modelleren. Diagrammen zijn abstract en worden vlug te complex. Met UWE kan men applicaties modelleren voor veel toestellen zoals Tables, TVs, ... Misschien zou het gemakkelijker zijn om een subset van UWE te definiëren enkel voor smartphoneapplicaties. De basis kan het UWE navigatie model zijn. Om de user interface te modelleren kan er misschien beter gezocht worden in de richting van een What-You-See-Is-What-You-Get editor.

REFERENCES

Contents

1 Introduction .................................................. 1

2 State Of The Art .............................................. 4
   2.1 Model Driven Web Engineering .......................... 4
      2.1.1 Model Driven Development ........................ 4
      2.1.2 Web Modeling Languages ........................... 5
      2.1.3 Conferences ....................................... 6
   2.2 Standards for Web Engineering ........................ 7
      2.2.1 Object Management Group .......................... 7
      2.2.2 Unified Modeling Language ........................ 7
      2.2.3 Object Constraint Language ......................... 11
      2.2.4 Model Driven Architecture ........................ 12
      2.2.5 QVT and Model2Text ............................... 13
   2.3 UML-based Web Engineering ............................. 13
      2.3.1 Overview .......................................... 13
      2.3.2 Requirements model (CIM) ......................... 14
      2.3.3 Content model .................................... 15
      2.3.4 Navigation model ................................ 15
      2.3.5 Process model .................................... 16
      2.3.6 Presentation model ................................ 17
      2.3.7 Viewpoint Synchronization ........................ 19
      2.3.8 Weaving in adaptivity with aspects ............... 20
      2.3.9 Flexibility of concerns ............................ 21
      2.3.10 UWE tools and Code Generation .................... 22
   2.4 Conclusion .............................................. 23
3 Specification

3.1 Vision

3.1.1 Mission

3.1.2 Target customers

3.1.3 Key Factors to Success

3.1.4 Key Technology

3.1.5 Crucial factors for applicability

3.2 System scenarios

3.2.1 Actors of the system

3.2.2 Use cases

3.3 Research Challenges

3.3.1 System restrictions

3.3.2 Goal

3.3.3 Key Technology

3.3.4 Academic values and technical difficulties

4 Design

4.1 PIM to PSM transformation

4.1.1 Overview

4.1.2 Platform Independent Model: UWE presentation models

4.1.3 Platform Specific Model: Android HTML

4.1.4 Algorithm definition

4.1.5 UWE Profile

4.2 PSM to Code transformation

4.2.1 Overview

4.2.2 Menu

4.2.3 Forms

4.3 The framework

4.3.1 MVC

4.3.2 System start

4.3.3 Add model type

4.3.4 Add transformation

4.3.5 Add model
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.6 Execute transformation</td>
<td>60</td>
</tr>
<tr>
<td>4.3.7 Implementation</td>
<td>61</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>67</td>
</tr>
<tr>
<td>5.1 PIM to PSM transformation in ATL</td>
<td>67</td>
</tr>
<tr>
<td>5.1.1 Overview</td>
<td>67</td>
</tr>
<tr>
<td>5.1.2 Models saved in XMI format</td>
<td>68</td>
</tr>
<tr>
<td>5.1.3 Algorithm implementation</td>
<td>68</td>
</tr>
<tr>
<td>5.1.4 Exporting to standalone application</td>
<td>77</td>
</tr>
<tr>
<td>5.2 PSM to Code transformation with Acceleo</td>
<td>77</td>
</tr>
<tr>
<td>5.2.1 Overview</td>
<td>77</td>
</tr>
<tr>
<td>5.2.2 Algorithm implementation</td>
<td>78</td>
</tr>
<tr>
<td>5.2.3 Stand alone JAR</td>
<td>84</td>
</tr>
<tr>
<td>5.3 The framework in Java</td>
<td>85</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>89</td>
</tr>
<tr>
<td>6.1 Application use case</td>
<td>89</td>
</tr>
<tr>
<td>6.1.1 Application</td>
<td>89</td>
</tr>
<tr>
<td>6.1.2 Transforming the application</td>
<td>90</td>
</tr>
<tr>
<td>6.1.3 Results</td>
<td>94</td>
</tr>
<tr>
<td>6.2 Transformation complexity</td>
<td>96</td>
</tr>
<tr>
<td>6.2.1 Execution time</td>
<td>96</td>
</tr>
<tr>
<td>6.2.2 RAM memory usage</td>
<td>96</td>
</tr>
<tr>
<td>7 Conclusions and outlook</td>
<td>98</td>
</tr>
<tr>
<td>7.1 Conclusion</td>
<td>98</td>
</tr>
<tr>
<td>7.1.1 Goals</td>
<td>98</td>
</tr>
<tr>
<td>7.1.2 Generating user interfaces with UWE</td>
<td>98</td>
</tr>
<tr>
<td>7.1.3 Working with EMF</td>
<td>99</td>
</tr>
<tr>
<td>7.2 Future work</td>
<td>99</td>
</tr>
<tr>
<td>7.2.1 Transformation definition diagrams</td>
<td>99</td>
</tr>
<tr>
<td>7.2.2 Versioning of metamodel</td>
<td>99</td>
</tr>
<tr>
<td>7.2.3 Transformation repository</td>
<td>100</td>
</tr>
<tr>
<td>7.2.4 UWE Navigation diagrams</td>
<td>100</td>
</tr>
</tbody>
</table>
8 Appendix

8.1 UWE symbols ................................................................. 102
8.2 ATL transformation language ...................................... 103
8.3 ATL Implementation .................................................... 106
  8.3.1 Complete list of ATL Helpers ................................ 106
  8.3.2 Leaf rules ............................................................... 110
8.4 Acceleo Implementation ................................................ 113
  8.4.1 componentModule.mtl ............................................. 114
  8.4.2 menuModule.mtl ..................................................... 115
  8.4.3 selectionModule.mtl ............................................... 116
Chapter 1

Introduction

The last decades tons of new devices have been introduced. PCs, laptops, PDAs, mobile devices, tablet computers, TVs,... They all run applications which need to be redesigned for each platform. Designing the applications for each platform has a high cost, since the programmer needs to have knowledge of different programming languages and platforms. Meanwhile the internet has evolved from a static publishing medium to a dynamic and interactive multimedia platform. Distributed web applications include functionalities like GPS, camera, motion sensors,... They are adaptable to the user, the environment and the device. This trend implies some changes in the way that web applications are designed. Web applications have specific concerns and issues which need to be addressed more than in traditional software applications. Furthermore, new modeling techniques are being developed. They describe a structured way to model applications for different hardware and software platforms in the mobile environment.

A new World Wide Web

The internet has evolved in different stages. Originally it was a static content provider, the shallow web. Later it became possible to submit content to the web, allowing people to fill in forms or upload files. This is called the deep web, it allowed limited user interaction. Along with the rise of social media, the wisdom web emerged. Advanced user interaction became possible with blogs, social networks, video, multimedia, tagging, discussing,... The content generation of the web became user based rather than one author providing everything. Recently, different mobile devices have emerged, giving birth to the mobile web. In the mobile web, content needs to be distributed to a lot of different devices over the internet. Each device having its own
data presentation constraints as well as interaction constraints (e.g. touchscreen, stylus, keypad based, ...). For the sake of being complete, we mention the semantic web. It allows data not only to be sent, but also to be interpreted by computer devices. This internet evolution is essential to understand the design of web applications. When developing a method to design web applications, the architect should always keep in mind that web applications are distributed, cross-platform and have a user based content generation.

**Concerns and challenges**

Different concerns arise when developing web applications. To handle these concerns in the best way possible, they should be embedded in the application development process. The first concern is data representation of the problem domain. This concern does not differ much from the traditional software design concern. More specific concerns for web design methods are navigation, presentation and personalization. Most web applications provide some sort of view of the data, over which can be navigated. Modeling this navigation for multiple devices is a challenge. Presentation needs to be changed for each device and platform with a common presentation model. It is not the same since navigation, as navigation focusses on reachability of certain views in the application while presentation focusses on layout aspects. Important in modern web applications is the look-and-feel. It is one of the main factors by which applications are judged. Finally, adaptation is an important concern. Adaptation is based on different kinds of context awareness. Logical awareness provides the type of devices, version of the operating system, hardware and software specs, ... Physical awareness provides for example location. User awareness tracks the user’s behaviour, preferences, knowledge, disabilities, ... A lot of attention is being devoted to context awareness and adaptation of the application to the context. A clear overview of the different concerns for web applications is given in Figure 1.1.

**Goals**

In this thesis a model based web engineering method, UWE, is extended to solve some of the issues of designing cross platform web applications. Current model driven technologies are investigated to find out if they could be used to build a tool for UWE. Different technologies are combined in a proof-of-concept framework.
Outline

In Chapter 2 a state of the art overview of web engineering is given. Several standards and technologies are discussed. We give an overview of different web modeling methods and research conferences that have influenced web engineering in general. Model driven engineering is discussed with current modeling and transformation languages. Next, a specific web modeling language, UWE, is explained in detail. Recent research and papers are mentioned and areas of further research are discussed. From these conclusions, we draw up a vision for an UWE dedicated framework in Chapter 3. Some essential functionality for such a framework is discussed. We make a proof-of-concept by designing the framework, together with two UWE transformations in Chapters 4 and 5. Finally, an evaluation and conclusions are made for further research in Chapters 6 and 7.
Chapter 2

State Of The Art

This chapter gives a state of the art description of model driven web engineering. First, we elaborate on model driven engineering and web engineering. Theoretical concepts are explained and some web engineering methods are discussed in Section 2.1. Next, we give an overview of engineering standards used in web engineering and this thesis. Section 2.2 gives an overview of some commonly used standards. Third, we explain one particular method in more detail. Section 2.3 gives a description of UML-based web engineering. This chapter ends with conclusions for further research in Section 2.4.

2.1 Model Driven Web Engineering

2.1.1 Model Driven Development

In model driven development (MDD) or model driven engineering (MDE) designers tend to design models, representing the system they are building. Models are later (automatically) transformed to code. Transformations are defined between different models (model-to-model) and from models to text/code (model-to-text). There are several interpretations of model driven engineering, for example the interpretation of Object Management Group, discussed in section 2.2.4. Over the years, numerous web engineering methods have been based on the model driven principle. Taking into account the different concerns for web applications (data representation, navigation, presentation and personalization), web engineering methods have been designed. Each method puts different accents in its design trajectory. Some focus on data-intensive applications, others on the hypertext aspect or on the architecture rather than functionality. Web modeling languages can be categorized as data oriented, hypertext oriented,
2.1 Model Driven Web Engineering

An overview of different web modeling methods and influences is given in Figure 2.1. It indicates the origin and orientation of several methodologies.

2.1.2 Web Modeling Languages

An overview of the main characteristics of the most recent web modeling languages is given. **OOHDM** (Object Oriented Hypertext Design Method) focuses on four design activities: conceptual design, navigational design, abstract interface design and the implementation. In each of these phases, object oriented models are designed. OOHDM is no longer being developed, but many web engineering methods have concepts based on this method. **WebML** (Web Modeling Language) has its own notation, partially based on UML. The focus is on data intensive applications. WebML is model driven with structural, composition, navigation, presentation and personalization models. There is decent tool support and code generation by the WebRatio tool. It is one of the most commercially developed tools in the web engineering domain. **Hera** is used for the development of semantic Web Information Systems. It

---

**Figure 2.1**: History of web modeling languages

---

\(^1\) Unified Modeling Language, more about UML in section 2.2.2
is model driven with conceptual, integration and application models. **WebSA** (Web Software Architecture [2, 9]) focuses on the architecture of the web applications, assuming functionality can be modeled with traditional methods. It is MDA oriented with requirements, functional and architectural models. The biggest contribution of WebSA is the architectural model, it consists of subsystem models, web component configuration models and web component integration models. **OO-H** (Object Oriented Hypermedia [9]) is partially based on OO-paradigm with tool support by the Visual-WADE tool. **UWE** is completely based on a UML profile. It is MDA oriented with content, navigation, presentation, user and adaptation models. There is tool support with several small plugins.

### 2.1.3 Conferences

Many workshops and conferences are being organized in the web engineering domain. Some important conferences are listed below.

- **ICWE**: International Conference on Web Engineering (2000 - 2012)
- **ICMT**: International Conference on Model Transformations (2006 - 2013)
- **CaiSE**: International Conference on Advanced Information System Engineering (1988 - 2013)
- **ECMDA-FA**: European Conference on Model Driven Architecture - Foundations and Applications (2005 - 2009), now called
- **ECMFA**: European Conference on Modelling Foundations and Applications (2010 - 2012)
- **MDWE**: Model Driven Web Engineering Workshop (2004 - 2012)
- **AOSD**: International Conference on Aspect Oriented Software Development (1999 - 2013)

Conferences that no longer take place, but have played an important role for the basics of web engineering:
2.2 Standards for Web Engineering

After discussing some web engineering methods, we continue with discussing some engineering standards. These standards are used frequently in the development of web applications.

2.2.1 Object Management Group

Object Management Group (OMG), is an international, open, non-profit organization that has the goal to provide common standards for the world. OMG’s mission statement states the following:

"OMGs mission is to develop, with our worldwide membership, enterprise integration standards that provide real-world value. OMG is also dedicated to bringing together end-users, government agencies, universities and research institutions in our communities of practice to share experiences in transitioning to new management and technology approaches like Cloud Computing." [21]

Both the industry and the academic world are involved in OMG. The standards we will use are UML, OCL, MDA, MOF and QVT. They are all registered trademarks of OMG.

2.2.2 Unified Modeling Language

Unified Modeling Language (UML) is a modeling language to specify the structure, behaviour and architecture of systems. It is widespread in the software development industry and has numerous tools to support it. The latest official specification is UML 2.4. An overview of the most common UML diagrams is given in Figure 2.2 and will be further elaborated in the remainder of this section.
2.2 Standards for Web Engineering

Figure 2.2: Overview of UML 2.4 diagrams.

Figure 2.3: Example of a use case diagram.
2.2 Standards for Web Engineering

Use case diagrams

The first type of diagram we will need, is a use case diagram. Use case diagrams contain actors and use cases. Actors are entities that interact with the system. They can be related by inheritance. Furthermore, they are connected to use cases. Use cases are actions that actors can perform with the system. They can be related to each other, with include and extends relationships. Use cases in general can be stereotyped for further refinement in the problem domain\(^2\). An example of a use case diagram is shown in Figure 2.3. Actors include Web Customer, Authentication and Paypal. Use cases include View Items, Make Purchase, Checkout and Client Register. The use case diagram is stereotyped as a subsystem. One actor is stereotyped as a service.

Activity diagrams

Activity diagrams are used to model the dynamic behaviour of a system. In UML 2.4 activity diagrams we can find initial process nodes and final nodes. Other nodes represent actions in the process (rounded boxes) or objects used (rectangle boxes). Activities can have input and output pins, to model object flow between them. Actions are connected with arrows to indicate flows. When a decision is made, an arrow branches. A branch can have a guard condition. Activity diagrams can describe activities that occur simultaneously. To model this, forks and joins are used. An example of an activity diagram for an online shopping checkout is given in Figure 2.4. The process starts in the top left corner, with the initial node. The user can choose to browse items or search items. This is modeled with a branch. Furthermore, an activity flow is defined for viewing items and adding them to the shopping chart. The proceed to checkout activity (left lower corner) is modeled with a rectangle box that has an arrow in the left side. This means the activity can be activated at any moment. After this activity, the checkout node and the final node follow.

Class diagrams

A third important type of diagram is the class diagram. A class diagram is a structural diagram that models entities in the application. Classes have properties and can be related to other class by inheritance or ownership associations. An example of a class diagram is given in Figure 2.10. The class registrationData has five attributes of types string or integer. The class diagram

\(^2\)Stereotypes are explained later this section.
contains stereotyped classes, they are explained in next section.

UML profile

UML allows for extensions of the language by means of a profile. A profile is a lightweight way to adapt UML to a particular domain or platform. Profiles are made of stereotypes, tagged values and a set of constraints. It allows a more refined vocabulary to be used, specific to the current domain. An example of a stereotype definition is given in Figure 2.5. From a normal UML element, a stereotyped class *Computer* is derived. It has three tagged values, Vendor, CPU and Memory. An example of a diagram with stereotyped classes is given in Figure 2.6. The classes *Review*, *User*, *Conference* and *Paper* are stereotyped as *navigation class*. As discussed before, Figure 2.3 contains an example of an actor that is stereotyped as *Service*. UML profiles have been created for numerous domains. The advantage of a profile is that you can use the same UML modeling tool for modeling different application domains. UML makes extensive use of OCL, discussed in the next section.

---

**Figure 2.4:** Example of a UML 2.4 activity diagram.
2.2.3 Object Constraint Language

Together with UML, Object Constraint Language (OCL)\cite{21} is used. OCL is an expression language, to (among other things) define constraints on UML diagrams. Like UML it works regardless of the target implementation language. Constraints can be added in notes, an example is given in Figure 2.6. They describe the context in which they apply. Furthermore, there are three types of constraints: invariants, preconditions and postconditions. Invariants define a constraint that must always be valid, or the system is in a invalid state. A precondition is a condition that is checked before executing a method. A postcondition is a condition that must hold after executing a method. Invariants are defined on class attributes, while preconditions and postconditions are defined on methods.\cite{17} An example of an OCL constraint, added to a UML diagram in a note, is given in Figure 2.6. Since meta models are also defined in UML, OCL constraints are used to help define and extend meta models for specific domains.

Figure 2.6: Example of OCL constraint.\cite{13}
2.2 Standards for Web Engineering

2.2.4 Model Driven Architecture

Another standard put forward by OMG is the Model Driven Architecture (MDA). MDA is OMG’s interpretation of general model driven engineering (MDD), see Section 2.1. MDA states a general way to develop applications with model driven engineering. It focuses on three principles: a direct representation of the system through models, general standards and automation. MDA in general works with MOF compliant models. MDA defines three viewpoints on the system. The computationally independent viewpoint, platform independent viewpoint and the platform specific viewpoint. Specific views of the system from the perspective of the previous viewpoints (in other words: models) are named computationally independent models (CIM), platform independent models (PIM) and platform specific models (PSM). CIMs specify the environment and requirements of the system. A PIM focusses on the operation of the system, independent of any particular platform. A PSM combines the PIM with additional information about a target platform.

The key concept of MDA is transformations, see Figure 2.7. CIMs are transformed in PIMs. After being refined and marked, PIMs can be transformed in other PIMs, or PSMs. Finally, PSM are transformed to code. Transformations between models and to code are discussed in the next section.

Each engineering domain can create a software engineering method, following MDA. To do so, it has to define a number of MOF compliant CIMs, PIMs and PSMs, with transformations between them. The focus should be automatization. In the field of web engineering, several MDA methods have been defined, such as UWE [14], WebSA [2], WebML [4] etc.

Figure 2.7: Model Driven Architecture by OMG.

3MOF: Meta Object Facility. An OMG standard to which meta models comply.
2.2.5 QVT and Model2Text

QVT stands for Query, View and Transformations. It is a standard to define transformation between MOF compliant models. Transformations use metamodels of source and target model as an input. QVT exists of 3 languages. The two of importance are QVT-operational and QVT-relations. QVT-relations defines a relationship between two models. Relationships can be checked. If they do not hold, they are enforced by adjusting the source and target model. QVT-Operational defines uni-directional transformations of models. There are several implementations of QVT, for example ATL\(^4\). To transform models to code (or more general: text), OMG has the Model to Text standard. It defines a template based way of transforming models into programming code. One implementation of the model to text standard is the Acceleo project. Both ATL and Acceleo are incorporated in the Eclipse Modeling Framework Project (EMF). This project provides tool support for several OMG standards by means of Eclipse plugins.\(^{22, 23, 5}\)

2.3 UML-based Web Engineering

2.3.1 Overview

In the third part of the state of the art, we dig deeper in the details of one web engineering method. UML-based Web Engineering (UWE) is a software engineering approach to develop cross-platform web applications. The method is based on the standards UML, OCL and MDA, previously discussed. These standards widely accepted and have numerous tools to support them. The UWE approach stipulates a complete design life cycle for web-applications. It defines a number of models and model transformations according to OMG’s Model Driven Architecture (MDA).\(^{13, 11, 14}\)

An overview of UWE models and transformations is given in Figure 2.8. At the top level we have the requirements model which is a computational independent model (CIM, see 2.2.4). It consists of a number of UML use case diagrams and activity diagrams (see 2.2.2). The requirement model is transformed in a number of functional models and in an architectural model. These are both platform independent models (PIM, see 2.2.4).

Functional models include the content model, navigation model and the process model. The content model contains a representation of the application domain. It is modeled by a

\(^4\)ATL transformation language will be used in this thesis, see also http://www.eclipse.org/atl/
number of class diagrams and interaction diagrams. The process model contains a number of business processes, modeled in UML class diagrams and activity diagrams. After being refined, the content, requirements and process model transform into parts of the navigation model. This contains a number of nodes through which the user can navigate in the application. The navigation model consists of UML class diagrams.

The architectural and functional models are merged into a "Big Picture" model that is a big machine state diagram, which can be checked automatically.[7, 8] The big picture model is a platform independent model. This is transformed to platform specific models for each platform we want to target. PSMs can be generated for Java EE platform, .NET platform,... In this thesis we will create a PSM for browser based HTML code on Android devices. The rest of the section is structured as following. First, all the models are discussed in more detail. Then we highlight some different aspects of UWE: viewpoint synchronization, adaptivity, flexibility of concerns and tool support. We conclude with possible research that can be done in the future. A UWE modeling example is given in the evaluation section of this book, see Chapter 6. A list of all the UWE symbols is given in Section 8.1.

2.3.2 Requirements model (CIM)

Use Case Diagrams

The UML use case diagrams are extended with 2 stereotyped use cases: <<Browsing>> and <<processing>>. These are added because they apply more to web applications. The use cases are graphically presented with the UML profile notation, or simplified with just the symbol. The browsing use case represents pure navigation within the application. The processing use case represents workflow functionality.[10] From the browsing use cases, navigation classes are generated (see navigation model 2.3.4). From the processing use cases, process flow diagrams are generated (see process model 2.3.5).

Activity Diagrams

Each use case can be refined in an activity diagram. In the diagram, the workflow of the use cases are detailed using more UML stereotypes deducted from the UML states. Stereotyped actions include <<Display-Action>>, <<user-Action>> and <<system-Action>>. Display actions are used to display some user interface elements. User actions are used to model some user input. System actions define steps in the process where the system is processing something.
2.3 UML-based Web Engineering

2.3.3 Content model

The content model (PIM) represents the application domain. It uses classic UML class diagrams to model its elements. No extra UML extensions are added. From content classes that are marked as navigation relevant, navigation classes are generated. (See navigation model 2.3.4)

2.3.4 Navigation model

To build the navigation model (PIM), the stereotypes <<NavigationNode>> and <<NavigationLink>> are used. They are derived from UML classes and UML class links. The stereotypes <<Process Class>> and <<Process Link>> are added to embed processes in the navigation diagram. In a second step the navigation model is enhanced by a set of access structures to improve navigation. Stereotypes for Menu, Index, Query and Guided Tour are added.
• <<Menu>> models a choice between different navigation nodes.

• <<Index>> models a list of indices. When the link is followed, the chosen index is a unique id for the element that follows.

• <<Query>> models a search action of the application.

• <<Guided Tour>> models an entry point to an ordered set of navigation class instances, through which the user can go forward and backward.

![Navigation Diagram]

Figure 2.9: Navigation diagram

An example of a UWE navigation diagram is given in Figure 2.9. All the elements of the diagram are UML Classes that are stereotyped. Logout, Login and Register are system processes (<<process>>). MainMenu and MovieMenu are stereotyped <<menu>>. SearchMovie is stereotyped <<query>>. Other regular navigation nodes are Home, Movie, Person,...

2.3.5 Process model

The process model (PIM) is used to model business processes of web applications. It is first derived from the requirements model and then refined. There are two views to the process
2.3 UML-based Web Engineering

model, a structural view and a behaviour view.[12] The structural view consists of a number of UML classes stereotyped as $<<\text{Process Class}}>>$ that are derived from conceptual classes. Extra classes can be added if they are relevant to the process. Process classes model information used in the process, they do not model process states or steps. The instances of process classes are used in the execution of the process. An example of the structural process model is given in Figure 2.10. The behaviour view or process flow model contains a set of UML activity diagrams. Each process is a UML subactivity or call state. Inputs and outputs of the processes are presented as object flows with earlier defined process classes.

![Diagram of structural process model](image)

**Figure 2.10:** Structural process model.

2.3.6 Presentation model

The navigation model is transformed to the presentation model (PIM). This can be later refined (transformed) using a style guide.[8] Presentation models exist of two views, a structural view and a user interface view.[12] The structural view models how the presentation space is divided and which elements can be displayed in the same space. The user interface view (UI) defines a basic user interface. For the structural model, stereotyped classes are added and (possibly composite) diagrams are built. Structural stereotypes are:

- $<<\text{presentation group}}>>$ to model a presentation sub-structure.
• <<presentation class>> to model a logical fragment or unit the user interface is made of. A presentation class is always derived from a navigational class or process class.

• <<presentation page>> a specialization of presentation group, used to model the smallest presentation unit that can be presented to the user.[12]

User interfaces can contain basic UI elements. Currently, UWE proposes <<text>>, <<form>>, <<button>>, <<image>>, <<audio>>, <<anchor>>, <<collection>> and <<anchored collection>>. An example of a user interface is given in Figure 2.11.

![Presentation diagram]

Figure 2.11: Presentation diagram
2.3 UML-based Web Engineering

2.3.7 Viewpoint Synchronization

One problem that arises when working with different models, is model synchronization or viewpoint synchronization. A viewpoint is an area of concern of the system. A view is a representation of the system through the perspective of a viewpoint. Thus in UWE, the viewpoints are content, navigation, presentation, business processes, ... The views are the specific models: content model, navigation model, process model, ... Suppose we transform a conceptual class in a navigational class. Later the name of the conceptual class changes. What will happen to the name of the corresponding navigation class? There must be an automatic way to keep track of the correspondences between the models. In general, UWE model transformations are clearly unidirectional. If the navigation model changes, there is no feedback to previous models that were used to build it. Specification of the correspondences is a first step to bidirectional transformations. There are two ways of modeling correspondences, by using extensional models and by using intentional models. Extensional models define correspondences between specific elements of the models. The correspondences are expressed in UML 2, with abstract dependencies. Some predefined extensional correspondences for UWE are given in Figure 2.12. The correspondence sameName relates content classes with navigation and process classes.

Intentional modeling of correspondences is achieved by relating types of elements from different models rather than elements themselves. So here we are working at metamodel level. In UWE all anchors of the presentation model are related to a navigational link. Since intentional correspondences are abstract, a basic set of extentional correspondences can be instantiated from the intentional ones.

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Client</th>
<th>Correspondence type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content::Class</td>
<td>Navigation::NavigationClass</td>
<td>sameName</td>
</tr>
<tr>
<td>Content::Class</td>
<td>Navigation::ProcessClass</td>
<td>sameName</td>
</tr>
<tr>
<td>Content::Association</td>
<td>Navigation::Index</td>
<td>validIndex</td>
</tr>
<tr>
<td>Navigation::Link</td>
<td>Presentation::Anchor</td>
<td>validAnchor</td>
</tr>
<tr>
<td>BusinessProcess::UserAction</td>
<td>Presentation::Button</td>
<td>validButton</td>
</tr>
</tbody>
</table>

Figure 2.12: Examples of common correspondences in UWE.

Using intentional and extentional models, a synchronization mechanism can be built. All the correspondences are saved with a client, supplier and type of correspondence. The client is the element affected by the relation. The supplier is an element that provides the change because it is changed. When an element is adjusted, a correspondences is searched with that element as supplier. When found, the client class of the correspondence is adjusted, depending
on the type of correspondence. Because of the importance of models, viewpoint synchronization is of a growing concern to UWE and MDA in general. Their is a clear need for a complete definition of unidirectional correspondences in UWE. Even more research can be done on bidirectional transformations. Because of the lack of decent tool support, it is difficult to implement synchronization in current plugin based UWE tools.

2.3.8 Weaving in adaptivity with aspects

Adaptivity is becoming increasingly important in the web 2.0. Users want a personalized application, their favorite links first, their favorite content on the front page, integration with friends and social networks,... The application can model user properties such as user knowledge, background, preferences, interest,... or context properties such as location, platform, hardware properties, software properties, place, time,... The application can adapt to all these parameters. Adaptivity is a cross-cutting concern, occurring all over the system. It is therefore difficult to separate the concern and to model it in a separate model. In aspect oriented modeling an independent model is built and later woven into the other models. This model could be an adaptivity model or user model, both addressing cross-cutting concerns. This way, all adaptivity logic is contained in one model instead of being spread over the different models. Adaptation in UWE includes[1]:

- adaptive link ordering,
- adaptive link annotation,
- adaptive link hiding,
- adaptive link generation.

The UML metamodel is extended as in 2.13 to incorporate aspect oriented modeling. The point in the application where aspects are used is called a pointcut, it is the place where the weaving will later take place. The advice part is the refinement that is made to the system according to the aspect. Aspects are divided in two groups, modeling aspects and runtime aspects. Runtime aspects include for example linkAnnotationAspects, linkTraversalAspects and linkTransformationAspects.
2.3 UML-based Web Engineering

2.3.9 Flexibility of concerns

Due to the rapid expansion of the web domain there is a need for a structured way to add concerns to modeling methods. MDA intrinsically supports the separation of concerns, but what happens if new concerns are added to UWE? Adding concerns can have the following impact on a web-engineering method [19]:

1. Definition of new modeling elements to capture additional requirements.
2. Redefinition of the metamodel.
3. Adaptation of the development process.
4. Adaptation of the modeling process and code generation tools.

Three types of new concerns are distinguished: dependent concerns, replacement concerns and orthogonal concerns. A dependent concern has a relation to one or more other concerns. If you modify a modeling element of the concern, it will affect other parts of the model. A replacement concern replaces another concern. It can refine it to capture additional requirements, or totally replace a concern. In this case the metamodel needs to be extended with a new element or an...
element has to be replaced. *Orthogonal concerns* introduce new, independent concerns. They imply an extention of the metamodel and new transformations. To keep track of the latest modeling techniques, we can adapt the design method in three ways.\[19\]

1. Extend the metamodel, define new transformations and extend the modeling tool.

2. Merge the method with a part of another web modeling approach that covers the concern. This includes a merged metamodel, defining transformations between methods and an extention of the tools.

3. By use of a common metamodel for all web engineering methods.

**Conclusion**

There is work to be done on the implementation of these theoretical concepts. Proof-of-concepts can be created by merging different metamodels of web methods. Again the lack of decent tool support is an issue.

**2.3.10 UWE tools and Code Generation**

A few plugins have been developed to build UWE models. They support the developer by checking the models and automating transformations.

**UWE4JSF**

UWE4JSF is implemented as a set of plugins for Eclipse. It supports full code generation for Java Server Faces (JSF).\[15\] The latest version dates from 2007. An overview of how the tool works is given in Figure 2.14. UML models with UWE stereotype applications are first transformed to *UWE Source Models* that comply to a UWE Metamodel. In this form, the UWE models can be validated easier. The transformation is executed using ATL. After validation, the UWE Source Model is transformed to an JSF-PSM. This model is an instance of the JSF Metamodel. It is a platform specific model, aimed at JSF technology. The platform specific model is transformed to code using Java Emitter Templates (JET). This transformation creates files representing the web application. The jsp files are not run directly on the JSF framework, but on the UWE4JSF framework on top of it. This framework is designed to reduce the complexity of the generated code and the transformation rules.\[15\]
2.4 Conclusion

We can conclude that more research can be done in the area of UWE:

- Flexibility of the UWE method is improved by designing a flexible way to add concerns. A proof of concept can be designed, in which the UWE meta model is (in a standardized...
way) extended with new concerns. The impact on the tools and engineering method can be investigated.

• Adaptivity of applications, and how to model it with aspect oriented modeling. A specific tool can be built that weaves a separate adaptation and user model in the other UWE models.

• Code generation for the business processes, and their integration with the user interface.

• There is a clear need for an UWE dedicated case tool. Instead of a classic plugin to prove UWE concepts, it is time for a more powerful case tool. It should support the UWE design trajectory. Technically it should have a flexible way of adjusting the UWE metamodel, insert transformations, weaving models,...

• More proofs of concept can be made regarding user interface code generation. No PIM to PSM transformations for UWE have been written yet for a large number of devices and technologies.
Chapter 3

Specification

In this chapter we specify the system we want to build. First we define a clear vision for a UWE dedicated CASE tool (Section 3.1). Next, we define a minimal system with a number of use cases and scenarios (Section 3.2). In the last section of this chapter, we elaborate the research challenges of this work (Section 3.3).

3.1 Vision

One of the conclusions of the state-of-the-art was that UWE has a clear need for more tool support. This section discusses the commercial potential for such a tool. The ultimate goal is to develop a UWE modeling tool, supporting all UWE models and transformations. This way, we only need to model a multi-platform application once, which reduces design time and improves reuse of models. This chapter is further divided in mission, target customers, factors for success, technological factors and applicability factors.

3.1.1 Mission

The mission is to develop a UWE dedicated CASE tool. The developer uses our system to create and transform UML models. He uses a built-in model editor to create the models, and refine them. These models are transformed to other models and again refined in the editor. There is a model checker that checks the models semantically and provides feedback to the users. Finally the models are transformed to code. Different IDE projects are created containing the code generated for different target platforms. Our system might create an X-code project containing object-C code, a Netbeans project containing server side PHP code and an Eclipse project
containing Dalvik code. Together these projects represent a cross-platform web application. After generation, the developer continues this development in the platform specific IDE’s.

3.1.2 Target customers

The CASE tool will target software engineers, more specifically designers of distributed web applications. One problem they are faced with today is the big diversity of devices on which web applications have to run. UWE tends to postpone device-dependent decisions about design and implementation as much as possible. We target the developers of rather complex web applications; this is where model driven engineering has the bigger advantage. These users are further on referred to as developers, not to be confused with the developers of the CASE tool we are describing.

3.1.3 Key Factors to Success

The key factors to determine that we have a powerful CASE tool are:

- Intuitive implementation of the different models and model transformations
- Generation of large portions of the target application’s code
- Generic engine that supports transformations between models in general
- Flexibility in adding and/or replacing transformations
- Performance of generated applications

3.1.4 Key Technology

The CASE tool will be a clear model driven tool. For that reason an intuitively implemented design methodology with models, model transformations and diagram editor is vital. We tend to deliver a state of the art CASE tool. Key technologies are:

- Unified Modeling Language (UML) to build models. Important are UML diagrams, UML profiles, model and metamodel interchange formats, see also Section 2.2.2
- OCL, object constraint language to aide in building UML models, see also Section 2.2.3
- MDA with computationally independent models, platform independent models and platform specific models, see also Section 2.2.4
• ATL, a transformation language between models, see also Section 8.2.

• Acceleo, a model to text transformation language.

3.1.5 Crucial factors for applicability

Crucial factors for the applicability of the CASE tool are:

• Low learning curve for people who are used to working with UML.

• Supports the platforms that the user is targeting.

• Rapid development process.

• High percentage of code can be generated.

From the key factors of success and crucial factors for applicability, we derive a number of quality attributes that are important for the system:

• Modifiability, the user has the ability to add and remove transformations, models and metamodels dynamically.

• Usability of the user interface. The user interface has to be intuitive and lower the learning curve.

• Usability when performing transformations. Since transformations are a delicate job to execute, extensive feedback to the user is needed. This includes detailed error reporting, information about success or failure of user actions performed, information about model integrity, ...

3.2 System scenarios

We highlight the most important functionality of the system to be build. We do this by using use cases and scenarios (UML sequence diagrams).

3.2.1 Actors of the system

The system has one direct actor, a Developer of a web application. This developer loads or creates models, transforms them, refines them and generates code from them. However, we do mention other actors. Since not all models have to be created within the system itself, we
3.2 System scenarios

Figure 3.1: Transformation lifecycle, as it is being created and used by different actors.

can see other modeling tools as an external actor. The developer interacts with it, building his models. Furthermore, the transformations used in the system are built by an expert in the modeling methodology. We name this actor the UWE Specialist. The UWE specialist interacts with the Eclipse Modeling Framework (2.2.5), creating transformations. These transformations are loaded and used in our system. A simple sequence diagram is given, showing model and transformation lifecycles in Figure 3.1. Although not all actors are directly interacting with our system, we feel it is necessary to clarify in this way.

3.2.2 Use cases

First, we give some high level use cases of the system. As pointed out in Section 3.1.4 our goal is to build an inherently model driven tool. Therefore the user must be able to add, remove and edit models. We also want the tool to use independent transformations (see Section 3.1.5). So other use cases are adding and removing transformations. These transformations are between models that comply to certain metamodel definitions. Because different model types (e.g. UWENavigationModel and UWEPresentationModel) comply to the same metamodel definition (UWEMetamodel), a distinction has to be made between model types and metamodels. Different model types can have the same metamodel. Thus, one use case is to add model types. Last but not least, we want to be able to check models syntactically. All the use cases discussed are given
3.2 System scenarios

Figure 3.2: General use cases of the system in the use case diagram in Figure 3.2 and elaborated further in Table 3.1. The most important use cases are discussed in detail, with a sequence diagram to clarify the scenario.
Table 3.1: High level use cases

<table>
<thead>
<tr>
<th>ID</th>
<th>Use case name</th>
<th>Primary actor</th>
<th>Complexity</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add model type</td>
<td>Developer</td>
<td>Normal</td>
<td>High</td>
</tr>
<tr>
<td>2</td>
<td>Add model</td>
<td>Developer</td>
<td>Normal</td>
<td>High</td>
</tr>
<tr>
<td>3</td>
<td>Add transformation</td>
<td>Developer</td>
<td>Normal</td>
<td>High</td>
</tr>
<tr>
<td>4</td>
<td>Execute transformation</td>
<td>Developer and transformation</td>
<td>Complex</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Define a model flow</td>
<td>Developer</td>
<td>Normal</td>
<td>Medium</td>
</tr>
<tr>
<td>6</td>
<td>Perform a model check</td>
<td>Developer</td>
<td>Complex</td>
<td>Low</td>
</tr>
<tr>
<td>7</td>
<td>Create a model in model editor</td>
<td>Developer</td>
<td>Complex</td>
<td>Low</td>
</tr>
<tr>
<td>8</td>
<td>Refine or edit a model in model editor</td>
<td>Developer</td>
<td>Complex</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 3.2: Use Case 1: Add a model type.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Add model type</td>
</tr>
<tr>
<td>Date Creation</td>
<td>12/02/2013</td>
</tr>
</tbody>
</table>

| Actors | Developer |
| Description | The developer adds a new type of model to the modeling environment. He has to insert some information about the model type: the name, metamodel name and location. |
| Trigger | User clicks "add model type" in the user interface. |
| Preconditions | 1. Program is started  
2. Program is not executing a transformation  
3. Model type does not exist already |
3.2 System scenarios

<table>
<thead>
<tr>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A new model type is added</td>
</tr>
<tr>
<td>2. A (new) metamodel is added</td>
</tr>
<tr>
<td>3. Information is shown to the user concerning success or failure of the adding</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Normal Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The developer inserts the model type name, eg UWENavigationModel</td>
</tr>
<tr>
<td>2. The developer inserts the metamodel name, eg UML 2.0</td>
</tr>
<tr>
<td>3. If it is new metamodel, the developer browses for the metamodel location.</td>
</tr>
<tr>
<td>4. The system checks the metamodel.</td>
</tr>
<tr>
<td>5. The system returns that the metamodel is valid.</td>
</tr>
<tr>
<td>6. The developer clicks save.</td>
</tr>
<tr>
<td>7. Information is shown in the console, indicating the success.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system returns that there is an error with the metamodel.</td>
</tr>
<tr>
<td>This implies that the user cannot click save.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user can cancel any time during the process.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>High: adding model types is important for model driven engineering tools.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequency of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal: developers are not expected to add model types frequently.</td>
</tr>
</tbody>
</table>
3.2 System scenarios

Figure 3.3: Use Case 1: scenario in a sequence diagram.

Table 3.3: Use Case 2: Add a model.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Add a model</td>
</tr>
<tr>
<td>Date Creation</td>
<td>12/02/2013</td>
</tr>
<tr>
<td>Actors</td>
<td>Developer</td>
</tr>
<tr>
<td>Description</td>
<td>The developer adds a model that he wants to transform. This model is of some model type (with metamodel). This model can be imported or it can be created with the built-in model editor.</td>
</tr>
<tr>
<td>Trigger</td>
<td>User clicks add model in user interface</td>
</tr>
</tbody>
</table>
### 3.2 System scenarios

**Preconditions**

1. Program is started
2. Program is not executing a transformation

**Postconditions**

1. A model is added
2. Information is shown to the user concerning success or failure of the action.

**Normal Flow**

1. User clicks "add model"
2. User enters a model name
3. User selects model location
4. User selects type of model
5. User confirms
6. A message is shown in the console, indicating the success

**Alternative Flows**  
User can cancel anywhere during the process.

**Exceptions**  
The selected model is not emf compliant.

**Priority**  
High: Important in model driven engineering. How the model was created, is less important.

**Frequency of use**  
High: the developer is expected to add may different models.

**Assumptions**  
If the model was not created by this environment, we assume it is semantically correct.
Figure 3.4: Use Case 2: scenario in a sequence diagram.

Table 3.4: Use Case 3: Add a transformation.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Add a transformation</td>
</tr>
<tr>
<td>Date Creation</td>
<td>12/02/2013</td>
</tr>
<tr>
<td>Actors</td>
<td>Developer</td>
</tr>
<tr>
<td>Description</td>
<td>The developer adds a possible transformation to the system.</td>
</tr>
<tr>
<td>Trigger</td>
<td>User clicks “add transformation” in the user interface</td>
</tr>
</tbody>
</table>
| Preconditions | 1. Program is started  
2. Program is not executing a transformation |
| Postconditions | A transformation is added to the system and information is shown to the user concerning success or failure of the action. |
3.2 System scenarios

<table>
<thead>
<tr>
<th>Normal Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. User clicks add transformation</td>
</tr>
<tr>
<td>2. User inserts transformation name</td>
</tr>
<tr>
<td>3. User indicates if the transformation is PIM to PIM, PIM to PSM or PSM to code.</td>
</tr>
<tr>
<td>4. User inserts transformation input model type.</td>
</tr>
<tr>
<td>5. User inserts transformation output model type (possibly code project)</td>
</tr>
<tr>
<td>6. User inserts transformation JAR</td>
</tr>
<tr>
<td>7. <strong>User adds input parameters</strong></td>
</tr>
<tr>
<td>8. User saves configuration</td>
</tr>
<tr>
<td>9. A message is shown in the console, indicating the success or failure of the action.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exceptions</strong></td>
</tr>
<tr>
<td>User clicks cancel anywhere during the process. In that case nothing is added.</td>
</tr>
<tr>
<td><strong>Priority</strong></td>
</tr>
<tr>
<td>High: adding transformations is important in a model driven tool.</td>
</tr>
<tr>
<td><strong>Frequency of use</strong></td>
</tr>
<tr>
<td>Normal: the developer is not expected to add many transformations. Once a transformation is added, it is used many times.</td>
</tr>
</tbody>
</table>
Figure 3.5: Use Case 3: scenario in a sequence diagram.

Table 3.5: Use Case 4: Execute a transformation.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Execute transformation</td>
</tr>
<tr>
<td>Date Creation</td>
<td>12/02/2013</td>
</tr>
<tr>
<td>Actors</td>
<td>Developer</td>
</tr>
<tr>
<td>Description</td>
<td>User connects a model with a transformation and executes it.</td>
</tr>
<tr>
<td>Trigger</td>
<td>User clicks transform in the user interface</td>
</tr>
</tbody>
</table>
### 3.2 System scenarios

<table>
<thead>
<tr>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Program is started</td>
</tr>
<tr>
<td>2. Program is not executing a transformation</td>
</tr>
<tr>
<td>3. Input model is semantically correct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Transformation has successfully executed.</td>
</tr>
<tr>
<td>2. A model or code is generated.</td>
</tr>
<tr>
<td>3. Information concerning transformation execution is shown in the console.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Normal Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. User inputs a model</td>
</tr>
<tr>
<td>2. User inputs a transformation</td>
</tr>
<tr>
<td>3. User inputs a target location</td>
</tr>
<tr>
<td>4. The system calls the JAR</td>
</tr>
<tr>
<td>5. The system executes the transformation</td>
</tr>
<tr>
<td>6. The system shows a message that the transformation is successfully executed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative Flows</th>
<th>If the system does not finish successfully, an error message is shown.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Model was not compliant to metamodel</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>High: important to prove the model driven tool.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Frequency of use</th>
<th>High. We consider this one of the most used use cases.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Notes and Issues</th>
<th>This is vital to the proof of concept</th>
</tr>
</thead>
</table>
3.3 Research Challenges

3.3.1 System restrictions

Building a model editor and a transformation framework that implements all the UWE transformations is a bit over-ambitious to do in one thesis. Therefore we have restricted our system in some ways. First we decided to leave out the model editor from the system. Various UML modeling tools already exist (MagicDraw, Enterprise Architect, ArgoUML,...) and from an academic point of view it would be pointless to build another one. Note the importance of UWE being UML based at this point. Secondly, we will not build every transformation of the UWE methodology. We will create two transformations. The first will be a PIM to PSM transformation, creating a PSM model from the UWE presentation model for android devices. Secondly we want to create a PSM to code transformation, implementing the PSM model with browser based HTML5 and Javascript. These two will provide a primitive proof of concept for the framework. Thirdly, no model checker will be implemented. This is out of the scope of a proof of concept.
3.3.2 Goal

We want to create a framework to support transformations between models and transformations from models to code. The system will be used by the developer to transform UML-models. It allows for dynamic adding of transformations. Two types of transformations can be added, Model-To-Text (M2T) and Model-To-Model (M2M). M2M and M2T transformations are loaded in the system with JAR files that are created in Eclipse Modeling Framework. For M2M, ATL is used as a transformation language. For M2T, Acceleo is used as a transformation language. The system allows for definition of different model types. When a model type is added, the developer adds a model type name and metamodel. There is a fixed list of metamodels but the user can add more dynamically. Note that different model types can have the same metamodel. Fixed transformations can be defined between model types, resulting in a transformation chain. When the developer executes a transformation, he has to enter the input and output model. The correct JAR is loaded and the transformation is executed. The system indicates if the transformation was a success or not. Output files are created on the location indicated by the user. These output files can be edited in the user’s modeling environment of preference, to be used in the next transformation step later. In this thesis we want to accomplish the following:

- Having built a working environment to incorporate the transformations. Both transformation should be reusable in other environments.
- To design and implement a PIM to PSM transformation for UWE. We want to build a generic transformation, taking any UWE presentation model as input. The transformation should create a new, syntactically correct model, without loss of modeled concepts or data.
- To design and implement a PSM to code transformation. The generated code should contain a set of user interfaces representing the model. The user interface should have their proper interfaces to the application logic.

3.3.3 Key Technology

Many of the standards and technology where discussed before. We will use UWE, which is based on UML, OCL and MDA. We will transform the models to other models using ATL, a QVT implementation. Models will be transformed to code using Acceleo.
3.3.4 Academic values and technical difficulties

In this work, we investigate, combine and extend some model driven engineering concepts. We envision encountering following problems:

- Working with models and metamodels definitions and interchange formats. There is no universally used way to export models and metamodels in a standardized format.

- Integrating different transformations in one coherent framework. The transformations need to be independent.

- Defining a platform specific model, aimed at HTML technology.

- The transformation algorithm to create the PSM UWE model from a PIM UWE model.
Chapter 4

Design

We split the design in three parts. First, we create a model to model transformation for UWE. In order to do so, we need to create a transformation algorithm between the metamodel elements of UWE and the metamodel elements of a platform specific metamodel. This algorithm will be implemented in ATL and used as an independent JAR. Second, we want to create a model to text transformation, transforming the PSM to code. It will be implemented using the Acceleo code generator and exported as an independent JAR. In a third part, we bring together the different types of transformations in a model driven framework. The ultimate goal is to use the framework and transformations we built to transform a UWE Presentation model to a platform specific presentation model and then to code.

Since the work is divided in three parts, we will split the design and the implementation in three parts as well. Section 4.1 will be about the PIM to PSM transformation. Section 4.2 explains the PSM to code transformation and Section 4.3 will be about the framework.

4.1 PIM to PSM transformation

4.1.1 Overview

In this section we will discuss the design of a model to model transformation. The transformation will be between a platform independent model (PIM) and a platform specific model (PSM). More about these types of models can be found in Section 2.2.4. Integrating a model to model transformation in the framework is important for the proof-of-concept. As noted before, the transformation must be standalone and independent of everything except its metamodels and input model. The transformation will be used to design applications with UWE. We want
to create a user interface from UWE Presentation models. These user interfaces will later be implemented in HTML, Javascript and CSS. Thus our PSM will be a UWE (presentation) model, aimed at HTML implementation language. Although HTML is platform independent, we will show that some implementation parts can be optimized based on characteristics and capabilities of specific mobile devices. In the rest of this section, we first give an overview of the PIM modeling elements. This means that some of the UWE metamodel elements will be discussed in more detail. We need this because the transformation we design will map PIM metamodel elements to PSM metamodel elements. In a second step, a PSM is defined. We argue why and how the PSM (meta)model is made. Next, the transformation itself is defined.

### 4.1.2 Platform Independent Model: UWE presentation models

In order to define a transformation, we must look at what elements we can encounter in the PIM. The presentation part of the UWE metamodel is given in Figure 4.1 (UWE profile version 2.1). We can divide the (presentation) metamodel in three categories: structural elements (presentationGroup), interactive elements (interactiveElement) and output elements (outputElement). These three groups are discussed in more detail.

**Structural elements (presentationGroup)**

Structural elements model structural aspects of the user interface. There are three extentions of presentationGroups: iteratedPresentationGroup, inputForm and presentationPages. Note that a presentationGroup element has an attribute navigationNode, indicating the navigation node it was derived from. This information can be used to get the content class needed to show data in the presentationGroup class. It can also be used to get semantic information about the element, which could be a menu, index or query. An iteratedPresentationGroup contains a number of elements with only one visible at once. When implemented, there is a navigation structure, to switch between the visible element. An inputForm is a structure to model input from the user towards the application. It contains a number of elements to input data and to output data. A presentationPage is a special case of presentationGroup, it is the top level and can’t be contained in other presentationGroups.
Figure 4.1: Part of the UWE metamodel, presentation diagram
Interactive elements (interactiveElement)

Interactive elements model interaction of the user with the system. This interaction can be to enter data (inputElement), pure navigation (anchor, tab) or activation of some logic (button). Important input elements are selection, textInput, imageInput and fileUpload. Selection models a choice between different elements. TextInput models an input structure for the user to enter text. ImageInput and fileUpload model input of image files and other files.

Output elements (outputElement)

Output elements are used to model the output of content. This content can be text, an image or other media (mediaObject). The content that is shown to the user might come from the content model. When implemented, these output elements need to load content from the content model implemented on the server. This would be modeled as follows. A class in the content model is created, from which a navigation class is derived\(^1\). From this navigation class, a presentationGroup is derived. This group is created in the presentation model. The elements of this presentation group might be output elements. So what will we do when we encounter an output element? We need to reverse the process that we just explained. First we look at the presentation element the output element is contained in. When this is derived from a navigation class, which is derived from a content class, we know how to set the output data.

Loss of semantic information

We notice a first problem in transformations. A navigation diagram might contain a menu. This menu has a number of other navigation classes\(^2\) it refers to. When we generate a presentation diagram from it, we need to keep a piece of information indicating what navigation node it was generated from. For a menu navigation node, a presentationGroup class will be generated with a number of anchors in it for the nodes the menu refers to. But when generating the user interface from this presentationGroup class\(^3\), the system needs to know it is generating a menu. The menu class from the navigation model was transformed to a presentationGroup class in the presentation model. Thus the input for the transformation is a presentationGroup and a number of anchors, which does not provide enough metadata. We need to know the navigationClasses

---

1. No content-to-navigation transformation is designed here. The magicUWE tool has a content-to-navigation transformation implemented.\[^3\]
2. navigation classes or nodes
3. The transformation we will build.
the anchors are referring to. For this transformation, we require the UWE models to keep as much semantic information as possible. In the case of the menu, we can do this by using the navigationNode attribute from the presentationGroup class.

4.1.3 Platform Specific Model: Android HTML

Having looked at the PIM elements, we now define a platform specific model. This model has to contain the same semantic information as the PIM. The modeled concepts and their meaning must not be lost in the transformation. An example of semantic information, is that a presentation group is a menu. In the final (HTML) implementation there will be a certain combination of DIV elements and Javascript functions forming the menu. Various parameters can be identified in implementing this menu. Functions implementing the dynamics can be implemented in different ways. Also the structure and layout of the menu can vary. An example of two possible menu implementations is given in Figure 4.2. The PSM metamodel will contain modeling concepts to identify a menu, but stay independent of the implementation parameters.

The model should on the other hand be useful to fluently generate HTML, Javascript and CSS code out of it. We say that the model is aimed at a target implementation language. Syntax elements of the target language might occur. We will not implement the UWE models in plain HTML. We will also generate layout with CSS and logic with Javascript. Therefore the PSM metamodel should be based on a combination HTML, CSS and Javascript concepts.

We define a new metamodel, combining the points discussed above. All the metamodel elements are described below. Modeling elements have a name. They can be abstract (italic) and extend other elements. They can be related to other elements with a certain multiplicity. Elements can have attributes. These attributes can be primitive types such as integer, string or boolean. They can also be of another element type. Finally, we define a number of enumerables. An overview of the PSM metamodel is given in Table 4.1.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>Relations: children (0..*) : component</td>
</tr>
<tr>
<td>bold</td>
<td>Extends: finalElement</td>
</tr>
<tr>
<td>button</td>
<td>Extends: formElement</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Attributes:</td>
<td></td>
</tr>
<tr>
<td>◦ action : String</td>
<td></td>
</tr>
<tr>
<td>◦ autofocus : String</td>
<td></td>
</tr>
<tr>
<td>◦ buttonType : buttonType_enum</td>
<td></td>
</tr>
<tr>
<td>◦ disabled : Boolean</td>
<td></td>
</tr>
<tr>
<td>◦ formenctype : formType_enum</td>
<td></td>
</tr>
<tr>
<td>◦ method : formType_enum</td>
<td></td>
</tr>
<tr>
<td>◦ target : target_enum</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>component</th>
<th>Relations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ children (0..*) : component</td>
<td></td>
</tr>
<tr>
<td>◦ dimension : dimension</td>
<td></td>
</tr>
<tr>
<td>◦ parent : component</td>
<td></td>
</tr>
<tr>
<td>Attributes:</td>
<td></td>
</tr>
<tr>
<td>◦ class : String</td>
<td></td>
</tr>
<tr>
<td>◦ content : String</td>
<td></td>
</tr>
<tr>
<td>◦ id : String</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>componentLoader</th>
<th>Extends: contentLoader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations: component : component</td>
<td></td>
</tr>
</tbody>
</table>

| contentLoader | Extends: function |

<table>
<thead>
<tr>
<th>css</th>
<th>Relations: html : html</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes: name : String</td>
<td></td>
</tr>
</tbody>
</table>

<p>| dataModel |
| dimension | Attributes: |
| ◦ Height : String |
| ◦ Width : String |</p>
<table>
<thead>
<tr>
<th>div</th>
<th>Extends: component</th>
</tr>
</thead>
<tbody>
<tr>
<td>finalElement</td>
<td>Extends: component</td>
</tr>
<tr>
<td>form</td>
<td>Extends: component</td>
</tr>
<tr>
<td></td>
<td>Attributes: form_id : String</td>
</tr>
<tr>
<td>formElement</td>
<td>Extends: component</td>
</tr>
<tr>
<td></td>
<td>Relations: form : form</td>
</tr>
<tr>
<td></td>
<td>Attributes:</td>
</tr>
<tr>
<td></td>
<td>○ formId : String</td>
</tr>
<tr>
<td></td>
<td>○ name : String</td>
</tr>
<tr>
<td>formHandler</td>
<td>Extends: function</td>
</tr>
<tr>
<td></td>
<td>Relations: form : form</td>
</tr>
<tr>
<td>formSubmitter</td>
<td>Extends: formHandler</td>
</tr>
<tr>
<td></td>
<td>Relations: button : button</td>
</tr>
<tr>
<td>formValidator</td>
<td>Extends: formHandler</td>
</tr>
<tr>
<td></td>
<td>Relations: fields (0..*) : formElement</td>
</tr>
<tr>
<td>function</td>
<td>Relationships: container : javascript</td>
</tr>
<tr>
<td></td>
<td>Attributes: name : String</td>
</tr>
<tr>
<td>head</td>
<td>Attributes: title : String</td>
</tr>
<tr>
<td>html</td>
<td>Relations:</td>
</tr>
<tr>
<td></td>
<td>○ body : body</td>
</tr>
<tr>
<td></td>
<td>○ head : head</td>
</tr>
<tr>
<td></td>
<td>○ layout (0..*) : css</td>
</tr>
<tr>
<td></td>
<td>○ logic (0..*) : javascript</td>
</tr>
<tr>
<td></td>
<td>Attributes: name : String</td>
</tr>
<tr>
<td>Class</td>
<td>Extends</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>hyperlink</td>
<td>finalElement</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>image</td>
<td>finalElement</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>imageLoader</td>
<td>contentLoader</td>
</tr>
<tr>
<td>inputBox</td>
<td>formElement</td>
</tr>
<tr>
<td>italic</td>
<td>finalElement</td>
</tr>
<tr>
<td>javascript</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>menu</td>
<td>component</td>
</tr>
<tr>
<td>menuEntry</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td></td>
</tr>
<tr>
<td>paragraph</td>
<td>finalElement</td>
</tr>
<tr>
<td>selection</td>
<td>formElement</td>
</tr>
</tbody>
</table>
| selectionLoader | Extends: contentLoader  
Relations: selection : selection |
|-----------------|-------------------------|
| selectionOption | Attributes:              
◦ name : String  
◦ value : String |
| webServiceModel | Extends: dataModel  
Attributes:              
◦ IP : String  
◦ port : String |
| buttonType_enum | Literals:                
◦ button : 0  
◦ reset : 1  
◦ submit : 2 |
| formenctype_enum | Literals:                
◦ application : 0  
◦ multipart : 1  
◦ text : 2 |
| formType_enum   | Literals:                
◦ get : 0  
◦ post : 1 |
4.1 PIM to PSM transformation

<table>
<thead>
<tr>
<th>target_enum</th>
<th>Literals:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>◦ blank : 0</td>
</tr>
<tr>
<td></td>
<td>◦ parent : 1</td>
</tr>
<tr>
<td></td>
<td>◦ self : 2 top : 3</td>
</tr>
</tbody>
</table>

An example of a PSM is given in Figure 6.5 (right part). PSMs contain a root node, model. This node contains a number of html elements. Each html element has a head, body, logic and layout. The logic will contain a number of function elements. The body node contains different component elements. Components are extended in various ways. One particular way is a menu. A menu element contains menuEntries. Another important component is a form, containing formElements. FormElements are extended by inputBox, selection,... The function element discussed earlier, can be extended as formValidator or formSubmitter. They are related to form elements. Many more PSM modeling elements are defined, for extensibility.

4.1.4 Algorithm definition

Now we are ready to define the algorithm between the two model types. The input of the algorithm is a UWE Model, meaning a UML model with UWE stereotypes applied to it. These stereotypes are discussed in Section 4.1.2. The output of the algorithm is a PSM. PSM modeling elements are discussed in Section 4.1.3. We give an overview of what input patterns are matched to what output patterns

<table>
<thead>
<tr>
<th>Input pattern from PIM</th>
<th>Output pattern from PSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>no pattern</td>
<td>model : modelOut</td>
</tr>
</tbody>
</table>

For each transformation a model element is created. This element will contain all other PSM modeling elements.
### 4.1 PIM to PSM transformation

<table>
<thead>
<tr>
<th>Input pattern from PIM</th>
<th>Output pattern from PSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>presentationPage : pageIn</td>
<td>Create:&lt;br&gt;  ○ html : htmlOut&lt;br&gt;  ○ body : bodyOut&lt;br&gt;  ○ head : headOut&lt;br&gt;  ○ div : divOut</td>
</tr>
<tr>
<td></td>
<td>Initialise:&lt;br&gt;  ○ modelOut.pages &lt;- htmlOut&lt;br&gt;  ○ htmlOut.name &lt;- pageIn.name&lt;br&gt;  ○ htmlOut.body &lt;- bodyOut&lt;br&gt;  ○ htmlOut.head &lt;- headOut&lt;br&gt;  ○ bodyOut.children &lt;- divOut</td>
</tr>
</tbody>
</table>

pageIn has a number of composite elements. Presentation pages cannot be included in other pages, they are roots in the structural presentation model. When transforming the children of pageIn, more elements will be created. These elements are added to divOut.children.

<table>
<thead>
<tr>
<th>Input pattern from PIM</th>
<th>Output pattern from PSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputForm : formIn</td>
<td>Create:&lt;br&gt;  ○ form : formOut&lt;br&gt;  ○ formValidator : validatorOut</td>
</tr>
<tr>
<td></td>
<td>Initialise:&lt;br&gt;  ○ formOut.id &lt;- formIn.name&lt;br&gt;  ○ validatorOut.form &lt;- formOut&lt;br&gt;  ○ submitterOut.form &lt;- formOut&lt;br&gt;  ○ htmlOut.logic.content &lt;- validatorOut</td>
</tr>
</tbody>
</table>

When transforming the children of formIn, they are added to formOut.children. The elements that can be validated are added to validatorOut.fields.
### 4.1 PIM to PSM transformation

<table>
<thead>
<tr>
<th><strong>Input pattern from PIM</strong></th>
<th><strong>Output pattern from PSM</strong></th>
</tr>
</thead>
</table>
| presentationGroup : pgIn, the type of pgIn.navigationNode is *menu* | menu : menuOut  
Child elements of pgIn are transformed to menuEntry elements and added to menuOut.menuEntry. |
| presentationGroup : pgIn, all other presentation groups that are not of any of the previous types. | div : divOut  
The children of pgIn are transformed and added to divOut.children. |
| text : textIn, the parent element of the text is a presentationGroup derived from a menu. | menuEntry : entry |
| button : buttonIn                                                | Create:  
- button : buttonOut  
- formSubmitter : submitterOut |
|                                                               | Initialise:  
- buttonOut.content <- buttonIn.name  
- submitterOut.button <- buttonOut  
- htmlOut.logic.content <- submitterOut |
| textInput : textInputIn                                          | create: inputBox : inputOut  
initialise: inputOut.name <- textInput.name |
| anchor : anchorIn                                                | hyperlink : hyperlinkOut |
4.1 PIM to PSM transformation

<table>
<thead>
<tr>
<th>selection : selectIn</th>
<th>selection : selectOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>All the child elements of selectIn are transformed to selectionOption elements and added to selectOut.options.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text : textIn, the parent element of the text is a selection.</th>
<th>create: selectionOption : optionOut initialise:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>◦ selectOut.options &lt;- optionOut</td>
</tr>
<tr>
<td></td>
<td>◦ optionOut.name &lt;- optionOut.textIn.name</td>
</tr>
<tr>
<td></td>
<td>◦ optionOut.value &lt;- optionOut.textIn.name</td>
</tr>
</tbody>
</table>

4.1.5 UWE Profile

For this thesis we have defined a small subset of the UWE profile to work with. Italic elements are abstract stereotypes. The attributes referred to are mostly UML modeling elements. This profile reflects the UWE metamodel given in Figure 4.1. Not all UWE metamodel elements are defined and not all the elements defined here will be used in the transformation. More is implemented for extensibility reasons. Table 4.3 gives an overview of the profile elements.

Table 4.3: PSM metamodeling elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uweModel</td>
<td>Attributes:</td>
</tr>
<tr>
<td></td>
<td>base.Model : Model</td>
</tr>
<tr>
<td>userModel</td>
<td>Extends: uweModel</td>
</tr>
<tr>
<td>contentModel</td>
<td>Extends: uweModel</td>
</tr>
<tr>
<td>navigationModel</td>
<td>Extends: uweModel</td>
</tr>
<tr>
<td>presentationModel</td>
<td>Extends: uweModel</td>
</tr>
</tbody>
</table>
### node

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ isHome : Boolean</td>
</tr>
<tr>
<td>◦ isLandmark : Boolean</td>
</tr>
<tr>
<td>◦ guard : String [0..1]</td>
</tr>
<tr>
<td>◦ dataExpression : String [0..1]</td>
</tr>
<tr>
<td>◦ base.Class : Class</td>
</tr>
</tbody>
</table>

### navigationClass

Extends: node

### accesPrimitive

Extends: node

### process

Extends: node

### menu

Extends: node

### index

Extends: accesPrimitive

### link

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ guard : String [0..1]</td>
</tr>
<tr>
<td>◦ selectionExpression : String [0..1]</td>
</tr>
<tr>
<td>◦ isAutomatic : Boolean</td>
</tr>
<tr>
<td>◦ base.Association : Association</td>
</tr>
</tbody>
</table>

### navigationLink

Extends: link

### presentationElement

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ base.Class : Class</td>
</tr>
<tr>
<td>◦ base.Property : Property</td>
</tr>
</tbody>
</table>

### uiElement

Extends: presentationElement

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ id : String [0..1]</td>
</tr>
<tr>
<td>◦ visibilityCondition : String [0..1]</td>
</tr>
<tr>
<td>◦ styleClass : String[0..1]</td>
</tr>
</tbody>
</table>
### 4.1 PIM to PSM transformation

| **valueElement** | Extends: uiElement  
|                  | Attributes: valueExpression : String [0..1] |
| **outputElement** | Extends: valueElement |
| **interactiveElement** | Extends: valueElement |
| **inputElement** | Extends: interactiveElement  
|                  | ◦ dataProperty : String [0..1]  
|                  | ◦ submitOnChange : Boolean |
| **textInput** | Extends: inputElement |
| **selection** | Extends: inputElement  
|                  | Attributes: multiple : Boolean |
| **text** | Extends: outputElement |
| **image** | Extends: outputElement |
| **mediaObject** | Extends: outputElement |
| **presentationGroup** | Extends: uiElement  
|                  | Attributes: navigationNode : node |
| **presentationPage** | Extends: presentationGroup |
| **inputForm** | Extends: presentationGroup |
| **iteratedPresentationGroup** | Extends: presentationGroup  
|                  | Attributes:  
|                  | ◦ dataExpression : String [0..1]  
|                  | ◦ itemVarName : String [0..1] |
| **anchor** | Extends: interactiveElement  
|                  | Attributes: target_Node : String [1..*] |
| **button** | Extends: interactiveElement |
| **presentationAlternatives** | Extends: presentationElement |

An example of a UML model with applied UWE stereotypes, is given in Figure 6.5. A PIM is built with UML model, UML Class and UML Property elements. These are stereotyped with some of the stereotypes defined above. Note that there is a difference between the model, and
the diagrams built with it. Two different UML diagrams might contain the same element from the model.

4.2 PSM to Code transformation

4.2.1 Overview

In the second part of the design, we will design the PSM to code transformation. The starting point is the PSM metamodel, described in 4.1.3. This metamodel is already aimed at the target implementation language, i.e., HTML/Javascript/CSS. Many of the (non abstract) metamodeling elements have the same name as html tags. In the rest of this section, we discuss the mapping between PSM modeling elements and code. PSM modeling elements are printed italic. HTML tags are printed in upper case. We will focus on two important structures, menus and forms. We generate code for these PSM elements. Code generation can be extended to the complete PSM. For a proof of concept, menus and forms should do.

The transformation should start with a model element, representing the application model. This contains a number of html elements. Each html element is mapped to an .html page with HTML tags. This HTML tag contains a HEAD and a BODY tag. In the HEAD, a TITLE tag is added. This information is loaded from the head relation of the html element. Furthermore for each of the associated css and javascript elements, a .css and .js file are created. In the HEAD section of the html page, these files are imported. For the body association of the html element, all the children are added recursively in the BODY tag generated before. All the children are of "type" component.

4.2.2 Menu

Our transformation can implement menus in various ways. Menus are modeled in the PSM with menu and menuEntry elements. (see 4.1.3) We still have some degrees of freedom to implement the menu. Therefore we consider the manner of implementing menus as a parameter for the transformation. One possible way could be using html unordered list tags (<UL>), another using div (<DIV>). Furthermore the layout can vary. One layout could list the menu entries, while another could show them as blocks. The layout of two menus is given in Figure 4.2.
4.2 PSM to Code transformation

4.2.3 Forms

A second focus of this transformation is forms. Forms are modeled with PSM form elements. We transform them to html <FORM> tags. Forms contain formElements, that can be extended as inputBoxes, selections,... In the PSM, these form elements can be linked to function elements. FormValidator elements are linked to a form, and to a number of form elements. FormSubmitter elements are linked to a form and a button. The exact implementation of these functions can vary. A possible simple implementation could be that the formvalidator gets the value of all the elements it needs to validate, and does some calculations with it. If the validation succeeds, it returns true, else it returns false. An implementation of a form submitter could call this validator, if it returns true, the form is submitted. Otherwise the form is not submitted. An example of generated Javascript code is given in Section 6.1.3. Two functions are generated for the register form: a validation function and a submit function. This is just one possible implementation for these function. We could refine the PSM even more. Suppose that we would want to submit the form with webservices. The element WSFormSubmitter extends formSubmitter. The WSFormSubmitter can have attributes that specify web service parameters, for more detailed modeling.
4.3 The framework

Part three is the framework. This needs to unify models and transformations in a design tool. As discussed in 3.3.1, we will not implement all the features of the tool. The most important use cases are implemented as a proof of concept. The framework is intended to be lightweight. There is no introspection of models and/or model transformations. From this point of view, there aren’t many technical difficulties to building the framework. The difficulty is in executing the transformation. To do this, the framework needs to work with an external transformation JAR in a way that is generalizable. We have designed the framework with a set of UML diagrams. In the first diagram, we discuss the different components of the application. This is translated in a class diagram. Interaction between classes is modeled with a set of sequence diagrams for the most important use cases. We would like to stress again that the user of this system is an application developer.

4.3.1 MVC

We chose to use a simple Model-View-Control architecture for the framework. The model contains project files. In this file, we store all information about models, metamodels and transformations. Since the framework is lightweight, we don’t store actual model data, but only references to where the models are stored. A component diagram is given in Figure 4.3. There are three main components, Persistence, Core and User Interface. The core contains the Models, Metamodels, Model types and Transformations. These classes contain references to other locations that contain the actual data. The persistence component contains classes that handle saving project data. Furthermore, we use this MVC structure to refine the scenarios in more details. The interacting elements are now the components of the system. Each use case is discussed in more detail.

4.3.2 System start

With the interaction diagram in Figure 4.4 we provide an overview of how the system starts. The user needs to perform a number of actions before starting to work on his application. The user starts the application by starting the Core. Startup procedure in the core creates a persistence module and a user interface. In the quality attributes of the system, we stressed the importance of feedback to the user of the system. Therefore we devote special attention to a log module, which is part of the user interface. Now the system is started. The user can load a project he
4.3 The framework

Figure 4.3: Components of the system.

is working on. When opening a project, an XML file is entered in the user interface. This file contains all project data. The core then activates the persistence to load in the project data. After all the data is read, the core refreshes the user interface. This reloads the data shown on the screen. Finally, feedback is shown to the user, indicating the project was successfully loaded. Note that a user can also create a new project instead of loading an existing one. This use case is not discussed. After loading a project, the user is ready to start his work. He can execute other use cases as discussed further on.

4.3.3 Add model type

Adding a model type use case is shown in Figure 4.5. This is based on use case 1, see Table 3.2. The user interacts with the user interface to open a window that allows him to add a model type. He enters the model type name. Next, he has to enter the metamodel. As explained before, model type is not the same as metamodel. This is because multiple model types can have the same metamodel (e.g., UML 2.0). In UWE, model types could be UWE Navigation model or UWE Presentation model. The user can choose to use an existing metamodel or to add a new one. When the user chooses an existing metamodel, he picks one from the list that is shown to him. When a new metamodel is added, the user enters the name and metamodel location. The
core should perform a check on the given metamodel. It should be a valid metamodel. If the metamodel is valid, it is saved. Saving the metamodel is shown in a separate interaction diagram in Figure 4.6. The user clicks save, which activates the core. The entered data is written to the persistence layer and synchronized with the XML. Feedback is given to the user to show that the saving was a success and the model type was created.

4.3.4 Add transformation

Adding a transformation is shown in Figure 4.7. It is based on use case 3, see Table 3.4. The user interacts with the user interface to open a window that allows him to add a transformation. This triggers the user interface to load possible model types from the persistence layer. These model types are shown to the user as possible input model types and output model types of the transformation. Next, the user enters the transformation name and input model type. The user chooses between a M2M or M2T transformation. When it is a M2M transformation, the user also enters the output model type. Finally, he picks the transformation location. This should be an external JAR. Upon saving, the JAR is checked. If it is a valid transformation, all the data is saved to persistence. Persistence writes the data to the project XML. When the JAR is not valid, feedback is given to the user with an error.

4.3.5 Add model

Adding a model is shown in Figure 4.8. It is based on use case 2, see Table 3.3. The user interacts with the user interface to open a window that allows him to add a model. The user can add a model by importing it from an external location, or choose a model created by the internal model editor. As mentioned in the system constraints, we will not implement a model editor, see 3.3.1. The user enters the location of the model on the drive. The user interface shows a list with possible model types. This list is loaded from the persistence. The user picks a model type for the model that he imported. Note that model types need to be defined in order to do so (previous use case).

4.3.6 Execute transformation

The final use case is executing a transformation, see Figure 3.6. This use case is shown in the interaction diagram of Figure 4.9. The user interacts with the user interface to open a window that allows him to execute a transformation. He selects the model that he wants to transform.
When doing so, the possible transformations for that model are loaded. The core gets all the transformations from the persistence. For each transformation it checks if it is applicable. The possible transformations are returned to the user interface. Next, the user selects one and executes the transformation. The transformation is executed by calling the external JAR with correct arguments. Feedback is returned to the user as to the success of the transformation. Note that transformations, model types and models must be added before executing this use case.

4.3.7 Implementation

The framework will be implemented in Java.
4.3 The framework

Figure 4.4: System start and loading a project.
4.3 The framework

Figure 4.5: Add model type.

Figure 4.6: Save model type reference.
4.3 The framework

Figure 4.7: Add transformation.
Figure 4.8: Add a model.
Figure 4.9: Execute a transformation.
Chapter 5

Implementation

This chapter discusses the implementation of the transformations and the framework designed in Chapter 4. The PIM to PSM transformation is implemented in ATL (Section 5.1). The PSM to code transformation is implemented with Acceleo (Section 5.2). The framework is implemented in Java (Section 5.3).

5.1 PIM to PSM transformation in ATL

5.1.1 Overview

The algorithm as described in Section 4.1.4 is implemented using ATL. ATL is a model to model transformation language, part of the Eclipse Modeling Framework (EMF). ATL implements the OMG QVT standard discussed in Section 2.2.5. It allows for creation of standalone applications able to perform a model to model transformation. Inputs for the transformations are two metamodels and an input model. In this case the input metamodels are the UWE profile (Section 4.1.5) and the PSM metamodel (Section 4.1.3). The input model is a UWE model. ATL is a declarative language, the transformation is defined by declaring transformation rules. First we run through the implementation of the algorithm, showing snippets of the implementation. Next, we discuss running ATL in an independent JAR. In Chapter 6 we analyse how the transformation’s execution time scales in function of the model size.
5.1 PIM to PSM transformation in ATL

5.1.2 Models saved in XMI format

UML models built in EMF are saved in XMI\footnote{XML Metadata Interchange Format} format. When stereotypes are applied, the file is internally structured in a special way. The UML profile is defined with a number of stereotypes. A stereotype extends a UML 2.0 element and has some extra attributes and relations. A model is saved in two parts. Suppose the stereotype \texttt{presentationPage} extends the UML \texttt{Class} element. UML Classes have attributes and can have composite classes inside them, which is standard UML. PresentationPage has an extra attribute: \texttt{navigationNode}, being the node it was derived from. When we create a presentationPage element in a model editor, with composite elements and \texttt{navigationNode}, this element is saved in two parts. The first part is saved as a UML Class element. This element contains references to the composite elements, together with all the classic UML attributes. The second part is stored separately: the presentationPage stereotype application. This element has an attribute \texttt{navigationNode} set and a reference to its \texttt{base Class}, the first part stored. This is a little confusing when we implement the algorithm, but with the right ATL helpers we can work with basic elements and stereotype elements in a convenient way.

5.1.3 Algorithm implementation

In order to have better understanding of the code snippets, we recommend the user to read Section 8.2 first. This section gives a basic explanation about the ATL transformation language. The full transformation code is given in appendix 8.3. First a number of helpers are defined, to aide handling UWE metamodel elements. One of the reasons for defining many helpers is explained above. We define helpers with context and helpers without context.

\textbf{Helpers without context}

We define following helpers without context:

\begin{verbatim}
helper def : pageCounter : Integer = 1;
helper def : formCounter : Integer = 1;
helper def : mainModel : PSMMeta!model = OclAny;
helper def : forms : Map(String,PSMMeta!form) = Map{};
helper def : pages : Map(String,PSMMeta!html) = Map{};
\end{verbatim}
PageCounter is a counter to index the number of presentation pages transformed. When no name is entered for a presentation page, this counter is used to form the name of the presentation page. FormCounter is used to count the number of forms in the application. When no name is entered for a form, this counter is used to form the name. A snippet of the form transformation rule is given here:

```plaintext
name : String = if(fClass.name.oclIsUndefined())
  then 'form'.concat(self.formCounter.toString())
  else fClass.name
endif;
```

MainModel is a PSM model element. This contains the complete application model. Every time a presentation page is transformed to an html element, it is added to this model. The transformation starts with creating this model, see next snippet. An entrypoint rule is unique and has no pattern matching section ('from' section). Setting the global variable mainModel is done by calling mainModel helper. In this case, self refers to the transformation module.

```plaintext
entrypoint rule start(){
  to
    mai : PSMMeta!model
  do{
    self.mainModel <- mai;
    self.debug('started');
  }
}
```

The forms helper is a map, mapping form names to the PSM form elements. When handling some elements of a form, a formSubmitter might be generated. This formSubmitter has an attribute form, the form it refers to. We can get the form element by using the form helper map. The pages helper is used to store the transformed pages with there name. When logic is generated to handle (parts of) a form, it needs to be stored in the logic attribute of the page containing the form. Using the pages map, we can get this element.

**Helpers with context**

A number of helpers are defined in the context of UML elements. This is because of the way profile application is saved. We define helpers on UML Element, UML Class and UML Property
elements. There are two main types of helpers, an example of each type is given.

```java
helper context UWEMeta!Element def : isOutputElement() : Boolean = 
    self.hasStereotype('text') or self.hasStereotype('image') or
    self.hasStereotype('mediaObject');
```

```java
helper context UWEMeta!Class def: getNavigationClass() : UWEMeta!navigationClass =
    UWEMeta!navigationClass.allInstances()->select(e | e.base_Class.equals(self))->first();
```

The helper isOutputElement checks if a certain element is a UWE output element. This is the case if stereotypes `text`, `image` or `mediaObject` are applied. This helper is based on OCL syntax. The second helper gets the stereotype application element of a UML element. Remember that the UML element and stereotype application are saved separately. To get the UML element from a stereotype application element, attributes such as `base_Model`, `base_Class` and `base_Property` are used.

### Rule flow

All the rules defined and the flows between them are given in Figure 5.1. In the activity diagram all actions represent rules implemented in ATL. A flow between rules indicates that a rule can be called from another rule. Decisions are made before calling rules. In many cases this decision is based on the type of an element. These types are determined using the helpers defined above. There are three matched rules: `start`, `presentationModel` and `navigationModel`. These rules do not require any input arguments, they look for a pattern in the UWE input model. E.g., the presentationModel rule is represented by:

```java
-- A rule for presentation models
rule presentationModel{
    from
        e : UWEMeta!presentationModel
    do{
        -- transform model
        self.model(e.base_Model);

        -- debug some stuff
    }
}
```
The pattern in the from section of the rule is matched. This implies that this rule counts for all presentationModel elements from the input model. It will select all the presentationModel stereotype application elements. This stereotype is applied to a UML model element. The elements selected by this rule thus have a reference to the UML model element, base_Model. This rule calls another rule to handle the transformation of the base_Model element, the model rule.

The rest of the rule flow goes as follows. Presentation models are picked out by presentationModel rule, and processed by the model rule. Navigation models are picked out by navigationModel rule. The focus of this transformation is the presentation model, so we pass the navigation models to the same model rule. Navigation models are not processed further. Note that for a full UWE transformation, the content model, the process model and the navigation model will also need to be transformed. The model rule further handles the presentation model. It looks at the packaged elements from the model. If they contain presentation pages, they are passed to the structuralElement rule. This is a general rule for all structural elements (see Section 4.1.2 structural element). The type of the input element is checked. Presentation pages are passed to presentationPage rule and input forms are passed to inputForm rule. When the type is presentationGroup, we first check if it was derived from a navigation node that is a menu. If so, the element is handled by menu rule. If not, the element is handled by presentationGroup rule. The presentationPage rule is given here:

```java
rule presentationPage(page : UWEMeta!presentationPage, pc : Integer, htmlPage : String){
   using
   {
      pClass : UWEMeta!Class = page.base_Class;
      name : String = pClass.name;
   }
   to
   css : PSMeta!css(name <- name),
   js : PSMeta!javascript(name <- name),
}
```
Figure 5.1: Rule flow
head : PSMMeta!head(title <- name),
comp : PSMMeta!"div"(),
body : PSMMeta!body(children <- comp),
html : PSMMeta!html(layout <- css, logic <- js,name <- name, head <-
head,body <- body)
do{
  -- perform details
  self.mainModel.pages <- html;
  self.pages <- self.pages.including(name,html);
  -- transform children
  for(p in pClass.getChildren())
  {
    if(p.isStructuralElement()){
      self.structuralElement(p,comp,name);
    }
  }
}

A number of PSM elements are created. An html element is created with css, javascript, head and body elements. All these elements are initialized and the page is added to the pages helper map. Next, the child elements are transformed. In this case, we only implemented one type of children, a structural element. When a child has this type, it is referred to the structuralElement rule. The presentationGroup rule is simple, it generates a PSM div element, and puts its children in it. The menu rule is given here:

rule menu(group : UWEMeta!presentationGroup,node : PSMMeta!component , htmlPage
  : String){
  using{
    bClass : UWEMeta!Class = group.base_Class;
    bMenuClass : UWEMeta!Class = group.navigationNode.base_Class;
    name : String = bMenuClass.name;
  }
}
This rule creates a PSM menu element. The elements of the menu need to be anchors stereotypes and passed to the `menuElementAnchor` rule. This rule simply creates a PSM menuEntry element. The `inputForm` rule is given here:

```atl
rule inputForm(form : UWEMeta!inputForm, parent : PSMMeta!component, htmlPage : String){
    using
    {
        fClass : UWEMeta!Class = form.base_Class;
        name : String = if(fClass.name.oclIsUndefined())
            then 'form'.concat(self.formCounter.toString())
            else fClass.name
        endif;
        html : PSMMeta!html = self.pages.get(htmlPage);
```
5.1 PIM to PSM transformation in ATL

to

psmForm : PSMMeta!form(id<-name,form_id<-name),
validator : PSMMeta!formValidator(form<-psmForm)
do
{
    self.forms <- self.forms.including(name,psmForm);
    html.logic->first().content <- validator;

    if(fClass.name.oclIsUndefined())
    {
        self.formCounter <- self.formCounter + 1;
    }

    parent.children <- psmForm;

    for(p in fClass.getChildren())
    {
        if(p.isStructuralElement())
        {
            -- There can not be presentation page in a form, or another
            --form in a form, so it has to be a presentation group
            self.presentationGroup(p.getPresentationGroup(),psmForm,htmlPage);
        }else if(p.isValuedElement()){ if(p.isOutputElement()){
            if(p.isTextElement())
            {
                self.text(p.getText(),psmForm,name);
            }else if(p.isImageElement()){
                self.image(p.getImage(),psmForm,name);
            }else if(p.isMediaObjectElement()){
                self.media(p.getMedia(),psmForm,name);
            }
        }else if(p.isStartElement()){
        }
    }
}
} else {
    self.debug('error in output element');
}

} else if(p.isInteractiveElement()){
    if(p.isAnchorElement()){
        self.anchor(p.getAnchor(), psmForm, name);
    } else if(p.isButtonElement()){
        self.button(p.getButton(), psmForm, name, htmlPage);
    } else {
        self.debug('error in interactive element');
    }

} else if(p.isInputElement()){
    if(p.isSelectionElement()){
        self.selection(p.getSelection(), psmForm, name);
    } else if(p.isTextInputElement()){
        self.textInput(p.getTextInput(), psmForm, name, validator);
    } else if(p.isImageInputElement()){
        self.imageInput(p.getImageInput(), psmForm, name);
    } else if(p.isFileUpload()){
        self.fileUpload(p.getFileUpload(), psmForm, name);
    } else {
        self.debug('error in input element');
    }
}

else{
    self.debug('an error occured multiplexing in a form');
}

self.debug('form transformed');

The rule creates a PSM form element and initializes it. A form validator is created and added to the html page its logic. Furthermore the rule multiplexes the child elements of the form and passes them to the correct rules. For the sake of clarity and extensibility, a separation is made between valued elements and structural elements. Amongst valued elements there three categories: output elements, interactive elements and input elements. This division is compliant to the UWE metamodel, see Section [4.1]. All the elements are referred to leaf rules. These rules do not call other rules anymore. The leaf rules are relatively simple, we do not discuss them. An overview of the leaf rules is given in Figure [5.1] and the full implementation is given in the appendix.

5.1.4 Exporting to standalone application

When designing an ATL transformation, there is a difference between working in Eclipse and using the transformation in a standalone application. The Eclipse environment performs an amount of work for the developer of the transformation. The most important is package importing and resolving URIs. The UWE metamodel is a UML profile based on UML 2.x. When running an ATL transformation with this metamodel (UWE Metamodel) in Eclipse, the EMF environment resolves the references to UML 2.x. The UML 2.x metamodel is saved in a JAR within Eclipse. When we export the transformation to a standalone application, we need to import these UML 2.x JARs manually. They will be re-packaged in our transformation JAR. Furthermore, a URI needs to be added to the environment manually. In the java code that activates the ATL transformation, we add the line of code given below. The UMLPackage Class is included in the imported UML JARs.

```java
Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
    UMLPackage.eNS_URI, UMLPackage.eINSTANCE);
```

5.2 PSM to Code transformation with Acceleo

5.2.1 Overview

The second part of the implementation is the PSM to code transformation. This transformation is implemented using Acceleo. Acceleo is part of the EMF and implements the Eclipse Model-To-Text standard. It is a template based transformation language. The input for this transformation

---

2Universal Resource Identifier
is a configuration file, the PSM metamodel as defined in Section 4.1.3 and an instance of it (a model). The output of the transformation is a number of files containing HTML, Javascript and CSS code. We discuss the implementation of the algorithm with templates. A number of templates are shown, for specific PSM metamodel elements. First, an overview of all the templates is given.

5.2.2 Algorithm implementation

Acceleo works with modules that define templates for certain metamodel elements. Each template is discussed in more detail. Figure 5.2 shows an activity diagram that describes the components of the transformation. There are four modules, one input model and two output objects. The generateModel module is the main module, the entrypoint for the transformation. This template looks for model elements in the PSM. It generates a Javascript file and a HTML file. Further child elements of the page are generated by the componentModule, this module takes component elements as input. It looks at the type of the component. If it is a selection or a menu, it is passed to the selectionModule or the menuModule. Otherwise code is generated in the HTML file. PSM component elements have children, which are passed recursively to the same componentModule.

Main module

The first template is the main template, where the transformation starts. The transformation will start by transforming a model element instance. The first part of the main template is given below:

```
[comment encoding = UTF-8 /
[module model('org.colpaert.mathias.PSMAndroid')/
[import org::colpaert::mathias::psm2android::common::componentModule/

[template public generateModel(aModel : model)]
[comment @main/]
```

The code between square brackets is Acceleo code. Everything else is code to be generated. In the second line the model type that will be transformed is given, this is named
5.2 PSM to Code transformation with Acceleo

Figure 5.2: Generation components and generated artifacts.

org.colpaert.mathias.PSMAndroid and is defined in Section 4.1.3. Next, other Acceleo modules are imported. In the main module only one other module is imported, the component module. The rest of the main module defines the template to transform a PSM model element. The first part is the HTML part, and goes as follows:

```plaintext
[for (h : html | aModel.pages)]
  [file (h.name.concat('.html'), false)]
<html>
  <head>
    <LINK REL=StyleSheet HREF="css/android.css" TYPE="text/css">
    [for (c : css | h.layout)]
      <LINK REL=StyleSheet HREF=["css/".concat(c.name)/].css" TYPE="text/css">
    [/for]
    <script src="js/jquery-1.8.3.js" type="text/javascript"></script>
    [for (j : javascript | h.logic)]
      <script src=["js/".concat(j.name)/].js" type="text/javascript"></script>
    [/for]
```
5.2 PSM to Code transformation with Acceleo

A model contains several HTML elements. For each *html* element, a page is created (second line). Some basic HTML tags are generated, `<HTML>`, `<HEAD>`, `<BODY>` and more tags within the head and body. The iteration variable in Acceleo that iterates over the pages is `h`, of type html (see Section 4.1.3). This contains relations *logic* and *layout*. For each layout element, an import of a css script is generated in the code. For each logic element, an import of a Javascript script is generated in the code. Last, basic jQuery libraries and default css files are imported. These libraries and default layout are later parametrised as inputs of transformation. In the body code, a `<DIV>` is generated, the header. This `<DIV>` contains a `<H1>` element with the title of the page. A second `<DIV>` is generated, containing the rest of the body code. The `h` element contains a relation to a *body* element that has multiple children. These are all of the abstract type *component*. Now the imported module component is used. For each body child we call the component module to generate code for it. The component template is discussed later, first we finish the main module. So far we generated a HTML file for *html* element of the *model*. Now we generate a Javascript file for each *logic* element of the *html*. The code is given below:

```html
[if not h.head.title.oclIsUndefined()]
<title>[h.head.title/]</title>
[/if]
<meta name="viewport" content="width=device-width; initial-scale=1.0; maximum-scale=1.0; user-scalable=0;" />
<meta name="HandheldFriendly" content="True" />
</head>
<body>
<div id="header">
<h1><a href="">[h.head.title/]"</a></h1>
</div>
<div id="body">
[for (com : component | h._body.children)][com.generateComponent(2)/][/for]
</div>
</body>
</html>
```

```plaintext```
```javascript
A model contains several HTML elements. For each *html* element, a page is created (second line). Some basic HTML tags are generated, `<HTML>`, `<HEAD>`, `<BODY>` and more tags within the head and body. The iteration variable in Acceleo that iterates over the pages is `h`, of type html (see Section 4.1.3). This contains relations *logic* and *layout*. For each layout element, an import of a css script is generated in the code. For each logic element, an import of a Javascript script is generated in the code. Last, basic jQuery libraries and default css files are imported. These libraries and default layout are later parametrised as inputs of transformation. In the body code, a `<DIV>` is generated, the header. This `<DIV>` contains a `<H1>` element with the title of the page. A second `<DIV>` is generated, containing the rest of the body code. The `h` element contains a relation to a *body* element that has multiple children. These are all of the abstract type *component*. Now the imported module component is used. For each body child we call the component module to generate code for it. The component template is discussed later, first we finish the main module. So far we generated a HTML file for *html* element of the *model*. Now we generate a Javascript file for each *logic* element of the *html*. The code is given below:

```html
[if not h.head.title.oclIsUndefined()]
<title>[h.head.title/]</title>
[/if]
<meta name="viewport" content="width=device-width; initial-scale=1.0; maximum-scale=1.0; user-scalable=0;" />
<meta name="HandheldFriendly" content="True" />
</head>
<body>
<div id="header">
<h1><a href="">[h.head.title/]"</a></h1>
</div>
<div id="body">
[for (com : component | h._body.children)][com.generateComponent(2)/][/for]
</div>
</body>
</html>
```
for (j : javascript | h.logic)
[file ('js/.' + j.name + '.js'), false]]

$(document).ready(function(){
    for (f : function | j.content)
        if f.oclIsKindOf(formValidator)
            function validate_[f.oclAsType(formValidator).form.form_id/]() {
                var validationResult = true;
                for (fe : formElement | f.oclAsType(formValidator).fields)
                    var [fe.name/] = $('#[fe.id/]').val();
                // Add your validation code here
                return validationResult;
            }
        [/if]
    [if f.oclIsKindOf(formSubmitter)]
        function submit_[f.oclAsType(formSubmitter).form.form_id/]() {
            var validationResult = validate_[f.oclAsType(formSubmitter).form.form_id/]();
            alert('submit not implemented yet');
            if(validationResult){
                // Add your submit code here
            }else{
                // Add error code here
            }
        }
    [/if]
}[/for]
[/for]
[/file]
Javascript elements contain functions (j.content). For each function we want to generate code. A file .js file is created in the folder js. The files are defined in the .ready function of jQuery, this means that the functions generated are dependent of jQuery and will not work without it. Since we only have two types of functions, we do not define separate modules for their code generation. In a fully fledged transformation, it would be recommended to generate a module for each metamodel element. The functions in the Javascript file are iterated by iterator $f$. $F$ can be of two types, formValidator and formSubmitter. Checking the type is based on OCL syntax. The name of the validation function is created with the id of the form it is linked to. In the function, a variable is defined, validationResult. Furthermore, all the fields of the validator are processed. They represent an element of a form that needs to be validated. We generated skeleton code that gets the value of these elements. It is now up to the user to implement more calculations with these variables to determine the validationResult.

The submitter function implements what to be done when a form is submitted. Upon generating this function, the name is formed with the id of the form its linked to. The submitter function calls the validation functions, to see if the form can be submitted. Further actions are left to the user. The following line of code links the submitter function to the form submit button:

```javascript
$('#\[f.oclAsType(formSubmitter).button.id/\]').click(function(){
    submit_\[f.oclAsType(formSubmitter).form.form_id/\]();
});
```

The .click function of the jQuery library is used. The jQuery selector is formed with Acceleo code that gets the id of the button. The code executed upon clicking the button is the submit function defined earlier. The result of generating a complete Javascript file might look like this:

```javascript
$(document).ready(function(){
    function validate_login_form(){
        var validationResult = true;
        var username = $('#login_form_username').val();
        var password = $('#login_form_password').val();
        // Add your validation code here
    }
});
```
return validationResult;
}

$('#login_form_log_in').click(function(){
    submit_login_form();
});

function submit_login_form(){
    var validationResult = validate_login_form();
    alert('sumit not implemented yet');
    if(validationResult){
        // Add your submit code here
    }else{
        // Add error code here
    }
}

componentModule

The component module transforms PSM component elements. We give three code snippets. The first is the initialisation, the second a code generation fragment and the third is the code to call other components. The following code initializes the module.

[module componentModule('org.colpaert.mathias.PSMAndroid')]

[import org::colpaert::mathias::psm2android::common::menuModule/]
[import org::colpaert::mathias::psm2android::common::selectionModule/]

[template public generateComponent(aComp : component,tabs : Integer)]
...

We make use of the PSMAndroid metamodel again. Furthermore, we import two other modules: menuModule and selectionModule. In this module code is generated for several elements of the PSM metamodel: div, form, inputBox, button, paragraph and hyperlink. A fragment of the code generation is given for div elements:
5.2 PSM to Code transformation with Acceleo

[if aComp.oclIsKindOf(div)]
<div>
[for (child : component | aComp.children)]
[child.generateComponent(tabs+1)/]
[/for]
[aComp.content/]
</div>

We don’t give the generation code for all the elements, the full code can be found in the appendix. The module calls two other modules in the following way:

[if aComp.oclIsKindOf(menu)][aComp.oclAsType(menu).menuGenerated(/)][/if]
[if aComp.oclIsKindOf(selection)][aComp.oclAsType(selection).generateSelection(/)]
[/if]

5.2.3 Stand alone JAR

As noted in Section 5.1.4 the Eclipse environment performs an amount of work for the developer of the transformation. The most important is package importing and resolving URIs. In this case, we need to import the PSMAndroid metamodel manually. This is done with the following line of code:

resourceSet.getPackageRegistry().put(PSMAndroid.PSMAndroidPackage.eINSTANCE.getNsURI(), PSMAndroid.PSMAndroidPackage.eINSTANCE);

With this, the Accleo code can be packaged in an executable JAR. The Eclipse environment automatically compiles the .mtl files with Acceleo code to .emtl files. We can do this compilation manually, or we can copy the .emtl files that where generated by Eclipse in the JAR. We copied the files in the JAR, to avoid the delicate manual compilation every time the transformation is executed. The created JAR can be run as follows:

java -jar jar-name.jar "locationPSMAndroidModel" "OutputFolderLocation" "propertiesFileLocation"
5.3 The framework in Java

As discussed before, the third part of the thesis implements a lightweight tool that brings models and transformations together. The tool was implemented following the design discussed in Section 4.3. We present some screenshots of the tool, following the use cases given in Chapters 3 and 4.

Starting system

First the user starts the system. A screenshot of the first screen is given in Figure 5.3. The concepts model, model type and transformation are important to the system. Therefore they have a custom, recurring icon. The user can view and add models, model types and transformations. When the system is started, the user can create or open a project. A project contains all information about an application a user is working in. It contains information about model types, metamodels, transformations and models. The tool is lightweight: instead of containing a metamodel definition, the project data will contain a reference to a file on the drive where the metamodel definition is saved. The same holds for transformations and models. When a project is created or loaded, the user can start to work.

Figure 5.3: Welcome screen
5.3 The framework in Java

Figure 5.4: Add model type

![Add Model Type]

<table>
<thead>
<tr>
<th>Name</th>
<th>Metamodel name</th>
<th>Metamodel location</th>
</tr>
</thead>
<tbody>
<tr>
<td>psm</td>
<td>androidPSMMetamodel</td>
<td>C:\Users\Mathias\workspaces...</td>
</tr>
<tr>
<td>UWEPresentation</td>
<td>UWE</td>
<td>C:\Users\Mathias\workspaces...</td>
</tr>
</tbody>
</table>

Figure 5.5: View model types

Use case 1: model types

As discussed in use case 1 (see Table 3.2), the user should be able to add model types. A screenshot of the user interface to add model types is given in Figure 5.4. The user is allowed to create a new metamodel for his model type, or use an existing one. Two model types with the same metamodel could be UWENavigation models and UWEPresentation models, which both have UWE Profile as metamodel. The user can view all the model types he added. A screenshot of the user interface to view model types is given in Figure 5.5.

Use case 3: transformations

Use case 3 describes that users should be able to add a transformation, see also Table 3.4. The user interface to add a transformation is given in Figure 5.6. The user enters the name of the transformation and picks the source model type. This model type was added before, following use case 1. The target of the transformation can be code or another model. Next, the user picks the location of the transformation executable. As noted before, the transformation must
5.3 The framework in Java

Figure 5.6: Add transformation

Figure 5.7: View transformations

be executed independently of the framework. A default output location is picked, in case of a model-to-model transformation, this should be a file. In case of a model-to-code transformation, this should be a folder. Next, the user should pick a configuration file. Picking a configuration file is optional. This functionality has been added, otherwise too many parameters needed to be passed when executing the JAR. Configuration files are essential to execute transformations in a dynamic way, but where not included in the design. The user can view the transformations he has added. A screenshot of this user interface is given in Figure 5.7.

Use case 2: models

Use case 2 describes that a user can add models to the system, see Table 3.3. As noted before, we will not build a model editor, so the models added will be made with other model editors. The user interface for adding models is given in the screenshot on Figure 5.8. The user picks a name for the model, and the model type. This model type was defined in use case 1. The location of the model is chosen, by clicking the location button. Next the model is added to the system. The user can view the models he has added. When he does this, the models are
ordered on model type. When he selects a model, the possible transformations are given, and they can be run by clicking the run button.
Chapter 6

Evaluation

6.1 Application use case

6.1.1 Application

We will use the tool made in the proof-of-concept, to generate parts of a small twitter-like application. In the application, the user can register and log in. After logging in, he can post messages on his page. The functionality is rather basic, but it should suffice for a use case. The application is modeled with the UWE methodology in the topcased modeling tool. The model of the application exists of a content model, a navigation model and a presentation model. No process model is defined. The three UWE models are given as stereotyped packages in Figure 6.1. There are only two classes in the content model, User and Message. Each user has a number of messages that he will see on his own wall. Furthermore, a user has a username and password. The presentation and navigation model are given in Figures 6.3 and 6.2.

The navigation model (Figure 6.2) contains five navigation nodes. Two processes, two navigation classes and a menu. The first node is the home node. It contains a menu that contains links to three other nodes. Two of them are processes, logging in and registering. The other is the wall containing the messages of the user. The presentation model (Figure 6.3) contains five classes. There are four presentation pages, a page to log in, a registration page, a home page and a wall page. The home page contains a presentation group, the mainMenu. These presentation pages are derived from the corresponding navigation node. The input of our transformation is the presentation model, so in order to know that mainMenu (stereotyped presentationGroup) is a menu, we must look for the corresponding node in the navigation model. The login and register page contain an input form. The main menu contains a number of anchors. The forms
6.1 Application use case

Figure 6.1: Application model structure

are further detailed in the diagram on Figure 6.4. The login form contains a username and a password field, with a button to log in. Furthermore you can log in as analyst or developer. In the register form, you have to enter username, password (twice) and the gender of the user (M or V).

There is a separation between the UML model built, and the way it is represented. The UML model represents the application, the diagrams are a number of views on the UML model. For example, the input forms occur in more than one diagram, but only one instance of them exists. A more strict representation of the UML model is given in Figure 6.5.

6.1.2 Transforming the application

Having designed this application, we want to generate as much code for it as possible using the framework and transformations we built. Screenshots and information on how to do so, can be found in Section 5.3. We give an overview of the steps needed to transform the model to code.

- Start the system and create a new project.

- Add UWEPresentation model type. A new metamodel is added, UWE Profile. This is the profile as discussed in Section 4.1.5 (see also Figure 5.4).
6.1 Application use case

Figure 6.2: Navigation model

Figure 6.3: Presentation model
6.1 Application use case

- **AndroidPSM model type.** A new metamodel is added, AndroidPSM. This metamodel is defined in Section 4.1.3.

- **Add a first transformation: M2M.** The JAR is the one built in Section 5.1. We enter the name, source model type (UWEPresentation), target model type (AndroidPSM) and the JAR location. When choosing the default output location, we choose a folder and create a .psmandroid file in it. Upon executing the transformation, this file (out.psmandroid) will be created. For adding a transformation, see also Figure 5.6.

- **We add the model of the Twitter application to the system** (see also Figure 5.8). We choose a name, UWEPresentation model type and import the model.

- **Execute the M2M transformation.** Output is shown in the output window.

- **Add out.psmandroid as a model.** The model type is AndroidPSM.

- **Add a second transformation: M2T.** The JAR is the one built in Section 5.2. We enter the name, source model type (AndroidPSM), target model type (code) and the JAR location. When choosing the default output location, we choose a folder where the code will be generated. Last, we pick a configuration file for the transformation. This file contains a number of parameters.

- **Execute the M2T transformation.** A number of HTML pages, Javascript files and css files are created.

---

**Figure 6.4:** Form layouts in more detail
Figure 6.5: Left: Strict UML model representation. Right: the platform specific model for the application.
6.1 Application use case

6.1.3 Results

When we transform the UWE model, we get a PSM. A screenshot of this PSM is given in Figure [6.5]. We can open this model with Eclipse. Upon creating an ecore metamodel, Eclipse allows for automatic generation of a model editor. The model created is aimed at HTML and Javascript. The register page contains a number of Javascript functions to validate and submit the register form. Note that the model is independent of exact implementation patterns. A menu element contains several menu entries, which can be implemented in various ways. The way this is implemented can be passed in the configuration file.

Furthermore, we take a look at the generated HTML pages. The main page is given in Figure [6.6]. The menu is implemented and optimized for android. The register page is given in Figure [6.6]. The modelled input boxes are generated. But the application is not ready to go yet. More html elements need to be implemented, in order to clarify the input form. We argue that this are lightweight tags as add-ons. The core of the application is the logic behind the form. This logic is given below:

Register.js

```javascript
$(document).ready(function(){
  function validate_register_form(){
    var validationResult = true;
    var username = $('#register_form_username').val();
    var password = $('#register_form_password').val();
    var password2 = $('#register_form_password2').val();
    // Add your validation code here

    return validationResult;
  }

  $('#register_form_log_in').click(function(){
    submit_register_form();
  });

  function submit_register_form(){
    var validationResult = validate_register_form();
    alert('Submit not implemented yet');
    if(validationResult){
```

```javascript
  // Further implementation code
}
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```

```javascript
});
```
6.1 Application use case

Figure 6.6: Generated pages. Left: the main page. Right: the register page.

// Add your submit code here

} else {
    // Add error code here
}

The validation form gets all the elements from the form. It is up to the developer to further implement what he wants the validation logic to do.
6.2 Transformation complexity

6.2.1 Execution time

We take a look at the complexity of the transformations built. We check how the transformation execution time scales with the size of the model. We use the twitter model as a reference for model size. Figure 6.7 shows the execution times of the model-to-model and model-to-text transformations in function of the model size. The size of the model transformed is given in the left graph. The execution times are given in the right graph. The transformations are evaluated on a PC with windows 8, 8GB RAM memory and an Intel i5 3.3 GHz core. The data is an average of 10 runs with an average standard deviation of 372 ms. Each ATL transformation creates a PSM model, which is transformed by the Accelo transformation. From these results we can conclude that we do not envision problems in executing times when the model is scaled. The ATL execution time goes up faster then the Accelo execution time.

6.2.2 RAM memory usage

Second, we look at the RAM memory used by the transformations. Figure 6.8 shows the memory usage of the transformations when executing 8 transformations. The first four executions (black) are ATL executions with input model sizes 1, 10, 50 and 100. The execution time goes up, but there is not more memory usage. The next four executions (purple) are Accelo executions with the PSM models created from the first transformations. The model sizes and execution times

Figure 6.7: Execution time in function of model size.
Figure 6.8: Memory usage for transformations.

can all given in Figure 6.7. We can conclude that there is no significant increase in memory usage when the model size scales up.
Chapter 7

Conclusions and outlook

7.1 Conclusion

7.1.1 Goals

We have reached the three main goals of the thesis, as described in Section 3.3.2. First of all, we have built a working environment in which we can incorporate transformations. The user can add transformations in a generic way. The transformations are independent and thus reusable in other environments. The second goal was to build a model-to-model transformation for UWE. This is to see how the framework works with models and model-to-model transformations. The transformation built, takes a UWE model and transforms it to a platform specific model. In order to do so, we defined a subset of the UWE profile and a metamodel for the platform specific model. The third goal was to create a small code generating transformation. A set of user interfaces are generated from the PSM. Upon implementing these three goals in a proof of concept, we can draw conclusions about the design of transformations and their use in a model driven design methodology.

7.1.2 Generating user interfaces with UWE

Having created a transformation for UWE presentation models, we can conclude some things about this. Modeling user interfaces with UML does not seem to be the most appropriate way. Diagrams tend to get overloaded. Furthermore, there needs to be a synchronization between the diagrams made and the strict UML model behind it, see also the note in Section 6.1.1. Other options to model user interfaces can be explored, more in the direction of a WYSIWYG\(^1\).

\(^1\)WYSIWYG: What You See Is What You Get
editor. UWE is a very broad modeling language. It can model applications for smartphones and laptops, but also for tvs and many more. This is very powerful but it is also a weakness because it is too broad. It might be an advantage to narrow the methodology down to just smartphone applications. One aspect of UWE with powerful potential is the navigation model. Navigation diagrams have a lot of possibilities to create hands-on tools with high quality generated code. In the future work we give a more detailed description, see Section 7.2.4.

7.1.3 Working with EMF

The Eclipse Modeling Framework provides a powerful basis for model driven engineering. It provides a simple way of defining a metamodel, with automatic generation of a model editor. Secondly, it provides good tools to define a UML profile. Model-to-model transformations and model-to-text transformations can be integrated fluently with these technologies. The only problem that arises here is that other modeling tools are not able to work with the UML formats Eclipse uses. The interchange of UML models, UML profiles and metamodels is still lacking in general.

7.2 Future work

7.2.1 Transformation definition diagrams

Model driven methodology has driven OMG to create standards for model-to-model and model-to-text transformation languages. There are several implementations available in the form of programming languages, such as ATL and Acceleo used in this thesis. The languages certainly can fulfill the goals of transforming models. But upon designing transformation, there seems to be a lack of tools to think on meta level. Defining an algorithm on high level requires careful planning. A mapping needs to be designed between metamodel elements, which is embedded in an algorithm. Putting thoughts on paper is difficult because no sort of diagram exists to do so. A customized UML diagram could be designed to do just this: design a model-to-model and model-to-text algorithm.

7.2.2 Versioning of metamodel

A powerful feature of the Eclipse tools (ATL, Acceleo) certainly is that it can be used independently of Eclipse. But when using a Profile as metamodel, JARs that define UML need to
be included. This raises the question: what happens to the transformation if UML migrates to a next version? The same question might be asked about the UWE profile defined. Models defined with a previous version of the UWE Profile will not work with the new version, even with the slightest adjustment. It could be worth thinking about how versioning of the metamodels can be incorporated in modeling tools and how it affects transformations. This would also help solving the problem of new concerns discussed in Section 2.3.9.

7.2.3 Transformation repository

The framework created, was designed to use transformations in an independent way. A logical step would be to completely decouple the transformations from the modeling tool. The tool could incorporate a transformation repository, that lists all the transformations and which type of models they transform. The transformation JAR can be migrated to a server, where they can be downloaded from or even executed on the server. The previous point discussed, versioning, could be merged with this.

7.2.4 UWE Navigation diagrams

This last point discusses a small vision for code generation from UWE navigation diagrams. Almost all applications have a number of user interfaces shown to the end-user. The user interacts with them. These interface are implemented differently, depending on the programming language used. For web applications this can be HTML/Javascript or native code. When the user interacts with the application, he switches from one view to the other. The idea is that the logic for this switching mechanism can be extracted from the rest. For example:

- Suppose an application has a menu, that is shown on almost every page. How will this be implemented? In a web application, each page that has the menu might contain a simple line of Javascript code `load("menu.html")` before the rest of the code of that particular page. The function load is defined as a general function that includes another file.

- Many applications might have a back icon on each page. When tapping/clicking it, you go back to the previous page. The logic that keeps track of where we are in the application and what page to go back to, can be separated from the rest of the application logic.

UWE navigation diagrams can be used as a basis to build web application navigation structure. Navigation code is generated from it. It should be possible to synchronize the model with
the code of the application. When the user adds new pages to the application, he adds new nodes in the diagram and connects them. The content contains parts that are generated (e.g. with javadoc @generated tags) and parts that are not touched by the code generator. In a more advanced situation, we can throw away the idea that each node should be a page. Nodes might be processes, and this processes can be executed on the device or on the server.
Chapter 8

Appendix

8.1 UWE symbols

A more complete explanation of UWE and its symbols can be found on the website\footnote{http://uwe.pst.ifi.lmu.de/}.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Stereotype</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>anchor</td>
</tr>
<tr>
<td>-</td>
<td>browsing use case</td>
</tr>
<tr>
<td>-</td>
<td>button</td>
</tr>
<tr>
<td>-</td>
<td>custom component</td>
</tr>
<tr>
<td>-</td>
<td>display action</td>
</tr>
<tr>
<td>-</td>
<td>display pin</td>
</tr>
<tr>
<td>-</td>
<td>external node</td>
</tr>
<tr>
<td>-</td>
<td>file upload</td>
</tr>
<tr>
<td>-</td>
<td>guided tour</td>
</tr>
<tr>
<td>-</td>
<td>index</td>
</tr>
<tr>
<td>-</td>
<td>image</td>
</tr>
<tr>
<td>-</td>
<td>image input</td>
</tr>
<tr>
<td>-</td>
<td>input form</td>
</tr>
<tr>
<td>-</td>
<td>interaction pin</td>
</tr>
<tr>
<td>-</td>
<td>iterated presentation group</td>
</tr>
</tbody>
</table>
8.2 ATL transformation language

ATL is a declarative model to model transformation language. It is an implementation of OMG’s QVT standard, integrated in the Eclipse Modeling Framework. ATL is firmly based on OCL. All OCL primitive types and functions can be used in ATL. Furthermore, ATL extends OCL with a number of operations. OCL types used frequently are Integer, String, Real, Boolean, ...
Helpers

ATL helpers are functions defined to aide in the transformation. ATL helpers are defined as follows:

```plaintext
helper [context context_type]? def : helper_name(parameters) : return_type = exp;
```

The `context` is the metamodel element on which the helper is defined. It is optional, helpers can be defined for the whole module. The helper has a number of `parameters`, the arguments for the function. Each argument has a name and type. Basic OCL types are used (Integer, Real,...) along with a number of extra types defined by ATL (Map, Tuple,...). The `return_type` indicates the return type of the function. `Exp` forms a declarative expression that is returned. An example of a helper is given in section 5.1.3.

Rule types

There are two types of transformation rules used in ATL: matched rules and called rules. **Match rules** match elements from a source model and generate elements of a target model. The pattern for generation in this rules is declarative. The structure of a matched rule is given below:

```plaintext
rule rule_name {
    from
        in_var : in_type [in model_name]? [(
            condition
        )]?[using {
            var1 : var_type1 = init_exp1;
            ...
            varn : var_typen = init_expn;
        }]?[to
        out_var1 : out_type1 [in model_name]? ( 
            bindings1
        ),
        out_var2 : distinct out_type2 foreach(e in collection)(
            bindings2
        )
    }
}
```
The \textit{rule\_name} indicates the name of the rule. It has a mandatory \textit{from} and \textit{to} section. The \textit{from} section contains the pattern that is matched in the input model. \textit{in\_var} is an example of the elements selected with type \textit{in\_type} according to condition \textit{condition}. We might use attributes of this source model element in the \textit{to} section. This attributes can be calculated in the optional \textit{using} section. Here, a number of expressions are built that can be used in the \textit{to} and \textit{do} section. The \textit{to} section is a declarative section that creates model elements. After all the \textit{to} sections from all the rules are executed, the imperative \textit{do} sections are executed. These contain a number of imperative statements. These statements can be a call to a \textit{called rule}. The structure of called rules is given below:

\begin{verbatim}
[entrypoint]? rule rule\_name(parameters){
    [using {
        var1 : var\_type1 = init\_exp1;
        ...
        varn : var\_typen = init\_expn;
    }]?
    [to
        out\_var1 : out\_type1 ( 
            bindings1
        ),
        out\_var2 : distinct out\_type2 foreach(e in collection)(
            bindings2
        ),
        ...
\end{verbatim}
out_varn : out_typen (  
    bindingsn  
)??
[do {
    statements
}]??
}

Called rules have input parameters. Furthermore, there can be one entrypoint rule. This rule will be executed by the module on transformation execution. Matched rules are always executed, but called rules (not entrypoint) are only executed when called. Logically, called rules do not have a pattern matching from section. The other sections, using, to and do are available and have the same function as in matched rules.

8.3 ATL Implementation

8.3.1 Complete list of ATL Helpers

-- Static helpers (global variables)
helper def : pageCounter : Integer = 1;
helper def : formCounter : Integer = 1;
helper def : mainModel : PSMMeta!model = OclAny;
helper def : forms : Map(String,PSMMeta!form) = Map{};
helper def : pages : Map(String,PSMMeta!html) = Map{};

-- Helper to check if an element has a certain stereotype
helper context UWEMeta!Element def : hasStereotype(stereotype : String) : Boolean =
    self.getAppliedStereotypes() -> collect(st | st.name) -> includes(stereotype);

    helper context UWEMeta!Element def : getRefElements() : Sequence(UWEMeta!Element) =
    UWEMeta!Element.allInstances() -> select(e | e.base_Class.equals(self));

-- DERIVED TYPE CHECKING HELPERS
helper context UWEMeta!Element def : isOutputElement() : Boolean =
self.hasStereotype('text') or self.hasStereotype('image') or 
self.hasStereotype('mediaObject');

helper context UWEMeta!Element def: isTextElement() : Boolean = 
self.hasStereotype('text');

helper context UWEMeta!Element def: isImageElement() : Boolean = 
self.hasStereotype('image');

helper context UWEMeta!Element def: isMediaObjectElement() : Boolean = 
self.hasStereotype('mediaObject');

helper context UWEMeta!Element def: isInputElement() : Boolean = 
self.hasStereotype('selection') or self.hasStereotype('textInput') 
or self.hasStereotype('imageInput') or 
self.hasStereotype('fileUpload');

helper context UWEMeta!Element def: isSelectionElement() : Boolean = 
self.hasStereotype('selection');

helper context UWEMeta!Element def: isTextInputElement() : Boolean = 
self.hasStereotype('textInput');

helper context UWEMeta!Element def: isImageInputElement() : Boolean = 
self.hasStereotype('imageInput');

helper context UWEMeta!Element def: isFileUploadElement() : Boolean = 
self.hasStereotype('fileUpload');

helper context UWEMeta!Element def: isInteractiveElement() : Boolean = 
self.hasStereotype('anchor') or self.hasStereotype('button');

helper context UWEMeta!Element def: isAnchorElement() : Boolean = 
self.hasStereotype('anchor');

helper context UWEMeta!Element def: isButtonElement() : Boolean = 
self.hasStereotype('button');

helper context UWEMeta!Element def: isValuedElement() : Boolean = 
self.isInputElement() or self.isOutputElement() or self.isInteractiveElement();

helper context UWEMeta!Element def: isStructuralElement() : Boolean = 
self.hasStereotype('presentationGroup') or self.hasStereotype('presentationPage') 
or self.hasStereotype('iteratedPresentationGroup') or self.hasStereotype('inputForm');

helper context UWEMeta!Element def: isPresentationPage() : Boolean =
self.hasStereotype('presentationPage');
helper context UWEMeta!Element def: isInputForm() : Boolean =
    self.hasStereotype('inputForm');
helper context UWEMeta!Element def: isPresentationGroup() : Boolean =
    self.hasStereotype('presentationGroup');
helper context UWEMeta!Element def: isNavigationNode() : Boolean =
    self.hasStereotype('node') or self.hasStereotype('navigationClass')
or self.hasStereotype('menu');
helper context UWEMeta!Element def: isMenu() : Boolean =
    self.hasStereotype('menu');

-- HELPERS TO GET STEREOTYPE ELEMENT FROM BASE CLASS ELEMENT
helper context UWEMeta!Class def: getAnchor() : UWEMeta!anchor =
    UWEMeta!anchor.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getButton() : UWEMeta!button =
    UWEMeta!button.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getButton() : UWEMeta!button =
    UWEMeta!button.allInstances()->select(e | e.base_Property.equals(self))->first();
helper context UWEMeta!Class def: getFileUpload() : UWEMeta!concreteElementType =
    UWEMeta!fileUpload.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getImage() : UWEMeta!image =
    UWEMeta!image.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getImageInput() : UWEMeta!image =
    UWEMeta!imageInput.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getIndex() : UWEMeta!index =
    UWEMeta!index.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getInputForm() : UWEMeta!inputForm =
    UWEMeta!inputForm.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getMenu() : UWEMeta!menu =
    UWEMeta!menu.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getMediaObject() : UWEMeta!menu =
    UWEMeta!mediaObject.allInstances()->select(e | e.base_Class.equals(self))->first();
helper context UWEMeta!Class def: getNavigationClass() : UWEMeta!navigationClass =
UWEMeta!navigationClass.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getNavigationLink() : UWEMeta!navigationLink =
UWEMeta!navigationLink.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getNavigationProperty() :
UWEMeta!navigationProperty = UWEMeta!navigationProperty.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getPresentationAlternatives() :
UWEMeta!presentationAlternatives = UWEMeta!presentationAlternatives.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getPresentationPage() :
UWEMeta!presentationPage = UWEMeta!presentationPage.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getPresentationGroup() :
UWEMeta!presentationGroup = UWEMeta!presentationGroup.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getSelection() :
UWEMeta!selection.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getText() :
UWEMeta!text.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Class def: getTextInput() :
UWEMeta!textInput.allInstances()->select(e | e.base_Class.equals(self))->first();

helper context UWEMeta!Property def: getTextInput() :
UWEMeta!textInput.allInstances()->select(e | e.base_Property.equals(self))->first();

-- Helper that checks if an element has an navigation node
helper context UWEMeta!Element def: hasNavigationNode() : Boolean =
not self.navigationNode.oclIsUndefined();

-- Helper to get navigation node
helper context UWEMeta!Element def: getNavigationNode() : UWEMeta!NavigationClass =
if(self.navigationNode.oclIsUndefined())
    then
        OclAny
    else
        self.navigationNode
endif;

-- Helper to get all children
helper context UWEMeta!Class def: getChildren() : Sequence(UWEMeta!Element) =
    self.nestedClassifier.union(self.ownedAttribute);

8.3.2 Leaf rules

Furthermore we give some of the leaf rules as discussed in section 5.1.3.

-- A rule to transform anchors
rule anchor(anchori : UWEMeta!Anchor, parent : PSMMeta!component, formId : String
    , htmlPage : String){
    using
    {
        aClass : UWEMeta!Class = anchor.base_Class;
        target : String = if(anchor.targetNode.oclIsUndefined())
            then
                'http://no.target.set'
            else
                anchor.targetNode.first().toString()
            endif;
        text : String = if(aClass.name.oclIsUndefined())
            then
                'hyperlink'
            else
                aClass.name.toString()
            endif;
8.3 ATL Implementation

```plaintext

} to

hyperlink : PSMMeta!hyperlink(target <- target, text <- text)
do{
    -- perform details
    parent.children <- hyperlink;
}

} to

-- A rule to transform buttons
rule button(button : UWEMeta!button, parent : PSMMeta!component, formId : String,
            htmlPage : String) {
    using {
        bClass : UWEMeta!Property = button.base_Property;
        target : PSMMeta!"target_enum" = 'self';
        text : String = bClass.name;
        html : PSMMeta!html = self.pages.get(htmlPage);
        id : String = formId.concat('_'+text.replaceAll(' ','_'));
    }
    to
    psmButton : PSMMeta!button(target <- target, content <- text,id <- id ,
                                name<-text),
    formSubmitter : PSMMeta!formSubmitter(button <- psmButton)
    do{
        -- perform details
        parent.children <- psmButton;
        formSubmitter.form <- self.forms.get(formId);
        html.logic->first().content <- formSubmitter;
    }
```

8.3 ATL Implementation

-- A rule to transform images
rule image(image : UWEMeta!image, parent : PSMMeta!component, loadId : String, htmlPage : String)
  using
  {
    iClass : UWEMeta!Class = image.base_Class;
    source : String = if(loadId.equals(''))
        'load from static link'
    else
        '',
    endif;
  }
to
  psmImgLoader : PSMMeta!imageSetter(),
  psmImage : PSMMeta!image(source <- source, dataSetter <- psmImgLoader)
do
  {
    -- perform details
    parent.children <- psmImage;
  }

-- A rule to transform selection elements
rule selectionEntry(text : UWEMeta!text, selection : PSMMeta!selection, formId : String)
  using
  {
    bClass : UWEMeta!Class = text.base_Class;
    name : String = bClass.name;
  }
to
option : PSMMeta!selectionOption(name <- name, value <- name)
do{
    selection.options <- option;
}
}

-- A rule to transform text inputs
rule textInput(inp : UWEMeta!textInput, parent : PSMMeta!component, formId: String, validator : PSMMeta!formValidator){
    using
    {
        sClass : UWEMeta!Property = inp.base_Property;
        name : String = sClass.name;
    }
to
    psmInput : PSMMeta!inputBox(name<-name,id <- formId.concat('_'+name))
do{
    -- perform details
    parent.children <- psmInput;
    if(not validator.oclIsUndefined()){  
    -- validate this field, this can be done depending on some parameters
    -- in the future
    validator.fields <- psmInput;
    }
}
}

8.4 Acceleo Implementation

The main code for the main module is given in section 5.2. We give the code for the other modules here.
8.4.1 componentModule.mtl

[comment encoding = UTF-8 ]
[module componentModule('http://org.colpaert.mathias.PSMAndroid')]

[import org::colpaert::mathias::psm2android::common::menuModule/]
[import org::colpaert::mathias::psm2android::common::selectionModule/]

[template public generateComponent(aComp : component,tabs : Integer)]
[if aComp.oclIsKindOf(div)]
<div>
[for (child : component | aComp.children)]
[child.generateComponent(tabs+1)/]
[/for]
[aComp.content/]
</div>
[/if]
[if aComp.oclIsKindOf(form)]
<div class="form">
<form name="[aComp.oclAsType(form).form_id/]">
[for (child : component | aComp.children)] [child.generateComponent(tabs+1)/]
[/for]
[aComp.content/]
</form>
</div>
[/if]
[if aComp.oclIsKindOf(inputBox)]
<input type="text" value="[aComp.oclAsType(inputBox).content/]"
id="[aComp.oclAsType(inputBox).id/]"
name="[aComp.oclAsType(inputBox).name/]"/><br/>
[/if]
[if aComp.oclIsKindOf(button)]
<button type="[aComp.oclAsType(button).buttonType.toString()/]"
8.4 Acceleo Implementation

[if aComp.oclAsType(button).disabled]disabled
[/if]method="[aComp.oclAsType(button).method.toString()]/"
id="[aComp.oclAsType(button).id/]"
name="[aComp.oclAsType(button).name/]">[aComp.content/]</button><br/>
[/if]
[if aComp.oclIsKindOf(paragraph)]
<p>[aComp.content/]</p>
[/if]
[if aComp.oclIsKindOf(hyperlink)]
<a href="[aComp.oclAsType(hyperlink).href/]"
target="[aComp.oclAsType(hyperlink).target.toString()]/">[aComp.content/]</a>
[/if]
[if aComp.oclIsKindOf(menu)] [aComp.oclAsType(menu).menuGenerated()]/[/if]
[if aComp.oclIsKindOf(selection)] [aComp.oclAsType(selection).generateSelection()]/[/if]
[/template]

8.4.2 menuModule.mtl

[comment encoding = UTF-8 ]
[module menuModule('http://org.colpaert.mathias.PSMAndroid')] 

[template public menuGenerated(aMenu : menu)]
<div class="menu" id="[aMenu.id/]"> 
<ul> 
[for (m : menuEntry | aMenu.links)] 
   <li><a href="[m.value/].html">[m.content/]</a></li> 
[/for]
   </ul>
</div> 
<div class="leftButton">menu</div> 
[/template]
8.4.3 selectionModule.mtl

[comment encoding = UTF-8 ]

[module selectionModule('http://org.colpaert.mathias.PSMAndroid')]

[template public generateSelection(aSelection : selection)]

<select id="[aSelection.id/]">
  [for (option : selectionOption | aSelection.options)]
    <option value="[option.value/]">[option.name/]</option>
  [/for]
</select><br/>
[/template]
Bibliography


List of Figures

1.1 Modeling aspects in UWE. ................................. 3

2.1 History of web modeling languages .......................... 5
2.2 Overview of UML 2.4 diagrams ................................. 8
2.3 Example of a use case diagram ................................. 8
2.4 Example of a UML 2.4 activity diagram ....................... 10
2.5 Stereotype definition ...................................... 11
2.6 Example of OCL constraint ................................. 11
2.7 Model Driven Architecture by OMG ........................ 12
2.8 Overview of UWE models and transformations ............... 15
2.9 Navigation diagram ...................................... 16
2.10 Structural process model .................................. 17
2.11 Presentation diagram ..................................... 18
2.12 Examples of common correspondences in UWE ............. 19
2.13 Extention of UML for AOD ................................. 21
2.14 UWE4JSF Generation Process .............................. 23

3.1 Transformation lifecycle, as it is being created and used by different actors .... 28
3.2 General use cases of the system ............................... 29
3.3 Use Case 1: scenario in a sequence diagram ................ 32
3.4 Use Case 2: scenario in a sequence diagram ................ 34
3.5 Use Case 3: scenario in a sequence diagram ................ 36
3.6 Use Case 4: Execute a transformation ........................ 38

4.1 Part of the UWE metamodel, presentation diagram ........... 43
4.2 Two menu implementations .................................. 57
4.3 Components of the system ........................................... 59
4.4 System start and loading a project ................................ 62
4.5 Add model type ....................................................... 63
4.6 Save model type reference .......................................... 63
4.7 Add transformation .................................................. 64
4.8 Add a model .......................................................... 65
4.9 Execute a transformation ............................................ 66
5.1 Rule flow .............................................................. 72
5.2 Generation components and generated artifacts ............... 79
5.3 Welcome screen ....................................................... 85
5.4 Add model type ....................................................... 86
5.5 View model types .................................................... 86
5.6 Add transformation .................................................. 87
5.7 View transformations ................................................ 87
5.8 Add a model .......................................................... 88
5.9 View models .......................................................... 88
6.1 Application model structure ........................................ 90
6.2 Navigation model ..................................................... 91
6.3 Presentation model ................................................... 91
6.4 Form layouts in more detail ......................................... 92
6.5 Left: Strict UML model representation. Right: the platform specific model for the application ........................................... 93
6.6 Generated pages. Left: the main page. Right: the register page .......................................................... 95
6.7 Execution time in function of model size ......................... 96
6.8 Memory usage for transformations ............................... 97
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>High level use cases</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Use Case 1: Add a model type</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Use Case 2: Add a model</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Use Case 3: Add a transformation</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Use Case 4: Execute a transformation</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>PSM metamodeling elements</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>PSM metamodeling elements</td>
<td>53</td>
</tr>
<tr>
<td>8.1</td>
<td>UWE Symbols</td>
<td>103</td>
</tr>
</tbody>
</table>