Vrije Universiteit Brussel

FACULTY OF SCIENCE
Department of Computer Science
System and Software Engineering Lab

Platform Ontologies for the Model-Driven Architecture

Dennis Wagelaar

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Science

Date: 07/04/2008
Promoters: Prof. Dr. Viviane Jonckers,
Dr. Ragnhild Van Der Straeten

Print: Flin Graphic Group, Oostkamp

© Dennis Wagelaar

© 2008 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. ++32 (0)2 289 26 50
Fax ++32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5487 482 9
NUR 992
Legal deposit D/2008/11.161/026

All rights reserved. No parts of this book may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author and the publisher.

Abstract

Software systems not only continue to grow more complex, but they are often required to run on multiple *platforms* as well. Common personal computer platforms are Microsoft Windows, Linux and Apple Mac OS X on a PowerPC or x86 hardware architecture. Hand-held devices present another range of platforms, such as Microsoft Windows Mobile, Qtopia/Embedix and Symbian running on an ARM or RISC hardware architecture. Each of these platforms look different from a software developer's point of view and requires the development of different software versions for each platform. This platform diversity makes it increasingly difficult to maintain software that is portable to multiple platforms. Software developers not only have to develop multiple software versions, but they also have to keep these versions synchronised and consistent in their common functionality.

In addition to this, platform technologies tend to *evolve*. When developing software for an evolving platform, software developers have to take into account that the users may use older versions of the platform. Developers may be confronted with the fact that their software is no longer compatible with an older version of the platform, because they do all their development and testing on the latest version of the platform.

If the current range of platforms can already be considered diverse, the vision of *Ambient Intelligence* only amplifies this diversity. Ambient Intelligence aims for a user-driven, service-based computing environment that includes personal devices as well as special-purpose embedded devices in the environment. The hardware and software combinations in such devices can vary widely.

The Object Management Group has acknowledged the problem of platform diversity by introducing the Model Driven Architecture (MDA). The MDA is centred around the use of software models. The software models provide a means to create partial, platform-independent software specifications that make use of platform abstractions. These abstractions are refined to platform-specific software models in a later stage of the development life cycle, using

model transformations. Currently, these model transformations implicitly assume a target platform for the platform-specific models. If other platforms must be targeted, new model transformations have to be created. This introduces a considerable maintenance burden for each additional platform we want to support.

It is possible to split up a model transformation into multiple refinement transformation steps, each of which introduces some partial platform dependencies into the software model. This makes it possible to reuse a refinement transformation for other platforms. It is not clear *when* we can reuse a refinement transformation, however, since the platform dependencies it introduces are still implicit.

When combining multiple refinement transformations for a target platform, most of the effort goes into checking that (1) the refinement transformations work together and that (2) they are executed in the right order. It is an extra burden to also (3) consider the platform dependencies that each refinement transformation introduces. One approach is to test the generated software on the target platform to tell if the software works on that platform. Testing on each platform is a time-consuming activity, however, and may even leave certain incompatibilities undetected until after deployment. Another approach is to use an automated configuration process that enforces the satisfaction of constraints, including platform dependency constraints. Such a configuration process does not exist for the MDA today.

We propose to use an explicit platform model, which serves as a vocabulary for describing platforms. This vocabulary is used as a basis to express platform instances as well as platform dependencies. By explicitly specifying the platform dependencies for each reusable refinement transformation, each transformation can be guaranteed as valid for a well-defined class of platforms. Because platform instances use the same platform model as a vocabulary, the platform model enables us to determine which platforms satisfy which platform dependencies. The platform model is expressed in the Web Ontology Language (OWL), which is an extensible language for describing ontologies. Ontologies are commonly used to represent domain knowledge and to provide a community of users with a controlled vocabulary. We use the OWL DL variant, which corresponds to *description logic* (DL) and allows us to apply automatic reasoning.

We also propose a configuration process for the MDA that is based on Software Product Lines (SPLs). Within the field of software engineering, most research on configuration has been conducted by the SPL community. SPLs integrate a number of software-intensive products that share a significant amount of functionality. As such, any software that is developed using the MDA approach can be considered as an SPL, since each platform-specific software product shares significant functionality with other platform-specific versions of that software product.

Samenvatting

Software systemen worden niet alleen steeds complexer, maar worden ook vaak vereist om op meerdere *platformen* te werken. Veel voorkomende personal computer platformen zijn Microsoft Windows, Linux en Apple Mac OS X op een PowerPC of x86 hardware architectuur. Draagbare apparaten vormen een bijkomend scala aan platformen, zoals Microsoft Windows Mobile, Qtopie/Embedix en Symbian draaiende op een ARM of RISC architectuur. Elk van deze platformen ziet er anders uit voor een software-ontwikkelaar en vereist de ontwikkeling van verschillende software-versies voor ieder platform. Deze diversiteit in platformen maakt het steeds moeilijker om software te onderhouden die overdraagbaar is naar meerdere platformen. Software-ontwikkelaars dienen niet alleen meerdere software-versies te ontwikkelen, maar zij moeten deze versies ook gesynchroniseerd en consistent houden wat hun gemeenschappelijke functionaliteit betreft.

Daarbij komt nog dat platform-technologieën vaak *evolueren*. Wanneer software-ontwikkelaars software schrijven voor een evoluerend platform, moeten zij er rekening mee houden dat de gebruikers weleens oudere versies van dat platform kunnen gebruiken. De ontwikkelaars kunnen hierbij geconfronteerd worden met het feit dat hun software niet langer compatibel is met een oudere versie van het platform, omdat het ontwikkelen en testen plaatsvindt op de nieuwste versie van het platform.

Als we het huidige scala aan platformen al divers vinden, dan wordt deze diversiteit alleen maar versterkt door de visie van *Ambient Intelligence*. Ambient Intelligence doelt op een door de gebruiker gedreven en op diensten gebaseerde computeromgeving, welke zowel persoonlijke apparaten als gespecialiseerde ingebouwde apparaten omvat. De hardware- en softwarecombinaties in zulke apparaten kunnen sterk variëren.

De Object Management Group heeft het probleem van platform-diversiteit onderkend door de introductie van de Model Driven Architecture (MDA). De MDA is opgebouwd rond het gebruik van softwaremodellen. De softwaremod-

ellen bieden een middel om partiële, platform-onafhankelijke software specificaties te maken die gebruik maken van platform-abstracties. Deze abstracties worden verfijnd naar platform-specifieke softwaremodellen in een later stadium van de software-ontwikkelingscyclus met behulp van *model transformaties*. Op dit moment gaan deze transformaties impliciet uit van een doelplatform voor de platform-specifieke modellen. Als er andere platformen ondersteund dienen te worden, moeten er nieuwe modeltransformaties gemaakt worden. Dit introduceert een aanzienlijke onderhoudslast voor ieder extra platform dat we willen ondersteunen.

Het is mogelijk om een modeltransformatie op te splitsen in meerdere stappen van verfijningstransformaties, waarbij elke stap enkele partiële platform-afhankelijkheden in het softwaremodel introduceert. Dit maakt het mogelijk om een verfijningstransformatie te hergebruiken voor andere platformen. Het is echter niet duidelijk *wanneer* we een verfijningstransformatie kunnen hergebruiken, omdat de platform-afhankelijkheden die zij introduceert nog steeds impliciet zijn.

Wanneer er meerdere verfijningstransformaties gecombineerd worden voor een doelplatform, gaat de meeste inspanning naar het controleren dat (1) de verfijningstransformaties samenwerken en dat (2) zij in de juiste volgorde worden uitgevoerd. Het is een extra last om ook (3) de platform-afhankelijkheden te beschouwen die elke verfijningstransformatie introduceert. Een mogelijke benadering is om de gegenereerde software te testen op het doelplatform om erachter te komen of de software werkt op dat platform. Het testen op elk platform is echter een tijdrovende bezigheid en laat mogelijk zelfs bepaalde incompatibiliteiten onopgemerkt tot na de installatie. Een andere benadering is om een geautomatiseerd configuratieproces te gebruiken dat het voldoen aan bepaalde beperkingen afdwingt, inclusief platform-afhankelijkheidsbeperkingen. Een dergelijk configuratieproces bestaat vandaag de dag nog niet voor de MDA.

Wij stellen voor om een expliciet platform-model te gebruiken, welke dient als vocabulaire voor het beschrijven van platformen. Dit vocabulaire wordt gebruikt als een basis voor het beschrijven van zowel platform-instanties als platform-afhankelijkheden. Door het expliciet beschrijven van de platform-afhankelijkheden voor elke herbruikbare verfijningstransformatie kan elke transformatie als geldig worden gegarandeerd voor een welgedefiniëerde klasse van platformen. Omdat platform-instanties hetzelfde platform-model gebruiken als vocabulaire, stelt het platform-model ons in staat om te bepalen welke platformen aan welke platform-afhankelijkheden voldoen. Het platform-model is uitgedrukt in de Web Ontology Language (OWL), wat een uitbreidbare taal is voor het beschrijven van ontologieën. Wij gebruiken de OWL DL variant, welke overeenkomt met *description logic* (DL) en ons toestaat om automatische redenering toe te passen.

Wij stellen ook een configuratieproces voor de MDA voor dat gebaseerd is op Software Product Lines (SPLs). Binnen het veld van software engineering is

het meeste onderzoek naar configuratie uitgevoerd door de SPL-gemeenschap. SPLs integreren een aantal software-intensieve producten die een aanzienlijke hoeveelheid aan functionaliteit gemeen hebben. Als zodanig kan alle software die ontwikkeld is met behulp van de MDA beschouwd worden als een SPL, omdat elk platform-specifiek softwareproduct een aanzienlijke hoeveelheid aan functionaliteit gemeen heeft met andere platform-specifieke versies van dat softwareproduct.

Acknowledgements

This is where I show my gratitude to all the people who made my Ph.D. possible. But before I start, I'd like to sketch some of the context in which I worked on my PhD. In August 2002, I moved from Losser, the Netherlands, to Brussels, Belgium to work on my Ph.D. at the System and Software Engineering Lab of the Vrije Universiteit Brussel (VUB). I quickly learnt that the VUB was very different from the University of Twente, where I had studied for my Master's degree. What I did not learn so quickly was how to integrate in such a different environment. Combined with my efforts to make life in Brussels work out, this took quite a bit of my time and attention. As a result, I'm finishing off this dissertation after almost six years in 2008.

The main hallmark of the VUB is that it takes its liberal stance seriously. I was really free in my choices, which also required a lot more independence on my part. Prof. Dr. Viviane Jonckers not only gave me the opportunity and freedom to pursue my own research topic, but she also had the patience for me to find my way through. I'd like to thank her for her confidence and support throughout these years. I hope to reward her confidence by closing off my Ph.D. period with this dissertation.

I would also like to thank Dr. Ragnhild Van der Straeten, Dr. Wim Vanderperren and Dr. Dirk Deridder for taking their time to discuss my research topic in detail with me. They have also read drafts of this dissertation in detail and provided me with valuable comments and directions.

I owe my gratitude to my Ph.D. committee members, for taking the time to read this dissertation in detail and for providing me with valuable comments. Apart from my advisors Viviane and Ragnhild, the committee members are Prof. Dr. Jean Bézivin, Prof. Dr. Yolande Berbers, Prof. Dr. Theo D'Hondt, Prof. Dr. Wolfgang De Meuter and Prof. Dr. Olga Detroyer.

I would also like to thank all of my colleagues and former colleagues, who have made my job more pleasant and were always there for interesting chats: Dr. Bart Wydaeghe, Dr. Wim Vanderperren, Davy Suvée, Dr. Bart Ver-

heecke, Miro Casanova, Dr. Ragnhild Van der Straeten, Dr. Maja D'Hondt, Dr. María Agustina Cibrán, Bruno De Fraine, Niels Joncheere, Mathieu Braem and my most recent colleagues Dr. Dirk Deridder, Andrés Yie, Mario Sanchez, Oscar González, Eline Philips and Dr. Andy Kellens. Special thanks go to Andres for test-driving my PlatformKit tool support and actually reading the manual. I would also like to thank Bruno De Fraine and Wim Vanderperren for managing the lab's servers with me. It provides an enormous freedom to be able to manage your own servers, but it is also good to know that I could share the responsibility with them. I really valued the unique situation in which our lab members contributed to our own servers.

Many thanks also go to the ATLAS team, with Jean Bézivin, Frédéric Jouault and Freddy Allilaire in particular, for sharing their ideas on model-driven engineering and letting me into the ATL community. Their open stance have allowed me to thoroughly experiment with model transformation and seeing my efforts fed back into the ATL (and AM3) tool. I hope for a fruitful collaboration in the future as well.

I'd like to thank my friends, Mark van Benthem, Joost Noppen, Frank Vlaardingerbroek and Tjim Wijering, for staying in touch after I've moved to Belgium. Even after not seeing each other for a long time, they have not forgotten about me. I would like to thank Joost in particular for even finding the time to proof-read my dissertation.

I would like to thank my parents, André and Paulien, who have always supported my studies from the very beginning. They weren't happy to see me leave to Belgium, but have never complained about it. They have in fact provided all the support they could for making my life in Belgium easier, so that I could spend more attention on my thesis. I would also like to thank Cynthia's parents, Philip and Frances, my brother Edwin and his wife Anja, as well as my uncle Gerard, for all their help with our house. If it weren't for our families, we wouldn't be living in our new house in Mechelen.

Finally, I'd like to thank my girlfriend Cynthia for being with me all this time. We moved to Belgium together in 2002 and we've also endured our common hardships here. Together, we've managed to find our place in Belgium. Cynthia has always supported me during this time and she kept a close eye on my thesis progress as well. Finishing this dissertation opens up our future to new opportunities, starting in our new home in Mechelen.

Dankwoord

Dit is waar ik mijn dank toon aan alle mensen die mijn doctoraat mogelijk hebben gemaakt. Maar voordat ik van wal steek, zou ik graag de context waarin ik aan mijn doctoraat gewerkt heb schetsen. In augustus 2002 ben ik verhuisd uit Losser, Nederland, naar Brussel, België, om aan mijn doctoraat te werken bij het Systeem en Software Engineering Lab van de Vrije Universiteit Brussel (VUB). Ik leerde al snel dat de VUB sterk verschilde van de Universiteit Twente, waar ik voor mijn ingenieursdiploma heb gestudeerd. Wat ik niet zo snel leerde was hoe te integreren in een zo verschillende omgeving. Samen met mijn inspanningen om mijn leven in Brussel in goede banen te leiden, nam dit een behoorlijk deel van mijn tijd en aandacht in beslag. Als gevolg leg ik de laatste hand aan dit proefschrift na bijna zes jaar in 2008.

Het hoofdkenmerk van de VUB is dat zij haar liberale standpunt serieus neemt. Ik was echt vrij in mijn keuzes, wat ook een stuk meer onafhankelijkheid van mijn kant vereiste. Prof. Dr. Viviane Jonckers gaf mij niet alleen de kans en vrijheid om mijn eigen onderzoeksonderwerp na te volgen, maar ze had ook het geduld om mij een weg te laten banen. Ik wil haar graag bedanken voor haar vertrouwen en ondersteuning gedurende deze jaren. Ik hoop haar vertrouwen te belonen door mijn doctoraatsperiode af te ronden met dit proefschrift.

Ik wil ook graag Dr. Ragnhild Van Der Straeten, Dr. Wim Vanderperren en Dr. Dirk Deridder bedanken voor hun tijd waarin zij mijn onderzoeksonderwerp in detail met mij hebben besproken. Zij hebben ook conceptversies van dit proefschrift in detail doorgelezen en hebben hebben mij voorzien van waardevolle commentaren en mogelijkheden voor verbetering.

Ik ben dank verschuldigd aan mijn juryleden die de tijd hebben genomen om dit proefschrift in detail te lezen en mij van waardevolle commentaren voorzien hebben. Naast mijn promotoren, Viviane en Ragnhild, bestaat mijn jury uit Prof. Dr. Jean Bézivin, Prof. Dr. Yolande Berbers, Prof. Dr. Theo D'Hondt, Prof. Dr. Wolfgang De Meuter en Prof. Dr. Olga Detroyer.

Ik wil ook graag al mijn collega's en voormalige collega's bedanken voor het aangenamer maken van mijn werk en het feit dat zij er altijd waren voor interessante gesprekken: Dr. Bart Wydaeghe, Dr. Wim Vanderperren, Davy Suvée, Dr. Bart Verheecke, Miro Casanova, Dr. Ragnhild Van Der Straeten, Dr. Maja D'Hondt, Dr. María Agustina Cibrán, Bruno De Fraine, Niels Joncheere, Mathieu Braem en mijn meest recente collega's Dr. Dirk Deridder, Andrés Yie, Mario Sanchez, Oscar González, Eline Philips en Dr. Andy Kellens. Een speciaal dankwoord gaat naar Andres voor het testen van mijn PlatformKit software en het daadwerkelijk lezen van de handleiding. Ik wil ook graag Bruno De Fraine en Wim Vanderperren bedanken voor het beheren van de servers op het lab samen met mij. Het verschaft een enorme vrijheid om je eigen servers te kunnen beheren, maar het is tegelijkertijd ook goed om te weten dat ik de verantwoordelijkheid kon delen met hen. Ik stelde de unieke situatie waarin de leden van ons lab bijdroegen aan onze eigen servers zeer op prijs.

Mijn dankbaarheid gaat ook naar het ATLAS team, met Jean Bézivin, Frédéric Jouault and Freddy Allilaire in het bijzonder, voor het delen van hun ideeën omtrent model-driven engineering en het feit dat zij mij binnengelaten hebben in de ATL gemeenschap. Hun open instelling heeft mij toegelaten om grondig te experimenteren met modeltransformatie en tegelijkertijd mijn inspanningen teruggekoppeld te zien in de ATL (en AM3) software. Ik hoop ook voor de toekomst op een vruchtbare samenwerking.

Ik wil graag mijn vrienden, Mark van Benthem, Joost Noppen, Frank Vlaardingerbroek en Tjim Wijering, bedanken voor het feit dat zij contact gehouden hebben nadat ik naar België verhuisd ben. Zelfs na elkaar gedurende lange tijd niet gezien te hebben, ben ik nog niet vergeten. Ik zou graag Joost in het bijzonder bedanken voor het feit dat hij zelfs de tijd heeft gevonden om mijn proefschrift na te lezen.

Ik wil graag mijn ouders, André en Paulien, bedanken voor het feit dat zij altijd mijn studie vanaf het begin ondersteund hebben. Zij zagen mij niet graag naar België vertrekken, maar hebben er nooit over geklaagd. Zij hebben daarentegen alle ondersteuning gegeven die zij konden bieden om mijn verblijf in België aangenamer te maken, zodat ik meer aandacht kon besteden aan mijn thesis. Ik wil ook graag Cynthia's ouders, Philip en Frances, mijn broer Edwin en zijn vrouw Anja, alsook mijn oom Gerard, bedanken voor al hun hulp met ons huis. Als onze familie er niet was, zouden we niet in ons nieuwe huis in Mechelen gewoond hebben.

Tot slot wil ik mijn vriendin Cynthia bedanken dat ze al deze tijd bij me is geweest. We verhuisden samen naar België in 2002 en we hebben hier ook onze gezamenlijke moeilijkheden doorstaan. Samen zijn we erin geslaagd onze plaats in België te vinden. Cynthia heeft mij tijdens deze periode altijd ondersteund en zij hield de vinger aan de pols waar het de voortgang van mijn thesis betrof. Het afronden van dit proefschrift opent onze toekomst voor nieuwe kansen, beginnend in onze nieuwe huis in Mechelen.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Today's software systems not only continue to grow more complex, but they are often required to run on multiple *platforms* as well. By *platform*, we refer to *the hardware and software combination on top of which our software runs*. Common personal computer platforms are Microsoft Windows, Linux and Apple Mac OS X on a PowerPC or x86 hardware architecture. Hand-held devices present another range of platforms, such as Microsoft Windows Mobile, Qtopia/Embedix and Symbian running on an ARM or RISC hardware architecture. Each of these platforms looks different from a software developer's point of view and requires the development of different software versions for each platform.

In the face of this platform diversity, it becomes increasingly difficult to maintain software that is portable to multiple platforms. Software developers not only have to develop multiple software versions, but they also have to keep these versions synchronised and consistent in their common functionality. Let's take Skype as an example, where the software developers have failed at keeping the functionality of their Windows, Mac and Linux versions in sync. The Windows version has all the latest features and, whereas the Mac version at least supports webcam, the Linux version has really fallen behind in the amount of provided features. Skype for Linux comes without webcam support and the developers are still working on a stable audio subsystem. The way that audio is accessed on Windows, Mac and Linux platforms is already so different, that there is no time left to address the differences in webcam access. The Skype developers are still far braver than others, since most software developers take the easy way out: they develop for one platform only. The problem is then simply passed on to the users who have to choose between platform X or Y, after which they are effectively locked into that platform.

Even if we look at a technology like Java, which was meant to overcome

platform differences ("Write Once, Run Anywhere"), we find that multiple Java versions exist today. Each of these Java versions are tailored towards specific usage scenarios and computing devices. To name a few: J2SE is the standard edition that is meant for desktop and laptop computers, J2EE is the enterprise edition that is meant for running server-side applications and J2ME is the mobile edition that is meant for resource-constrained and mobile devices. J2ME in turn is split up in separate versions for mobile phones, called Mobile Information Device Profile (MIDP)[1], and for PDAs, called Personal Profile (PP)[2].

The main difference between these Java versions lies in the libraries that make up the application programming interface (API). The API of J2ME MIDP for mobile phones is typically a stripped down version of the standard J2SE API. Functionality that is resource intensive and functionality that is not provided by the underlying platform is taken out of the API. New API elements are also added to J2ME MIDP to support functionality that is only relevant for mobile devices. Java reflection and the TCP/IP networking layer, for example, have been stripped from J2ME MIDP; Java reflection is too resource intensive and the TCP/IP networking layer is not (completely) available in the underlying operating system. In return, a special J2ME networking layer that can deal with limited TCP/IP networking has been added.

In addition to this, platform technologies tend to *evolve*. This evolution may involve just extending the API, but it can also involve an update of the programming language itself. In the case of Java, the language specification has been extended in J2SE 1.4 with *assertions* [GJJB00] and in J2SE 5 with *generics* and *annotations* [GJJB05]. When developing software for an evolving platform, software developers have to take into account that the users may use older versions of the platform. Developers may be confronted with the fact that their software is no longer compatible with an older version of the platform, because they do all their development and testing on the latest version of the platform.

If the current range of platforms can already be considered diverse, the vision of *Ambient Intelligence*, put forward by the Information Society Technologies Advisory Group (ISTAG) [DBS+01], only amplifies this diversity. Ambient Intelligence, or *AmI*, aims for a user-driven, service-based computing environment that includes personal devices as well as special-purpose embedded devices in the environment. The hardware and software combinations in such devices can vary widely[3]. It is also in such personal devices that another limitation of Java becomes apparent: the scope of Java as a platform abstraction layer is no longer complete. Deployment methods not only vary between dif-

[1] http://java.sun.com/products/midp/
[2] http://java.sun.com/products/personalprofile/
[3] http://www.comp.nus.edu.sg/~damithch/df/device-fragmentation.htm

ferent PDAs and mobile phones, but also between different network operators for the same mobile phone.

We have discussed how the Java technology often falls short in solving the platform diversity problem. Java is an example of a platform abstraction layer based on a virtual machine, where the software that runs on top of Java doesn't know about the details of the underlying platform. Other kinds of platform abstraction layers are script interpreters, standard libraries and frameworks. Regardless of the kind of abstraction layer, an abstraction layer solution to platform diversity will always be limited in the way that we have described for the Java technology. The causes of these limitations can be summarised as follows:

- Unreconcilable differences in underlying platform.
- Evolution of the platform abstraction layer.
- Scope of the platform abstraction layer.

In contrast to platform abstraction layers, which form an adapter between the software and the underlying platform, are solutions that adapt the software itself to the target platform. These solutions range from generative approaches that transform the software beforehand to reflective approaches that will react to platform differences at run-time. Each of these approaches have strengths and weaknesses. Generative approaches typically can't deal with post-deployment changes in the platform. Reflective approaches inevitably have some overhead in doing run-time platform checks and providing code that may never be executed on the target platform. In this dissertation, we will focus on a generative approach. We will discuss in chapter 8 how our thesis relates to reflective approaches.

The Object Management Group has acknowledged the problem of platform diversity by introducing the Model Driven Architecture (MDA). The MDA allows for "separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform." [MM03] and is centred around the use of software models. The software models provide a means to create partial, platform-independent software specifications that make use of platform abstractions. These abstractions are refined to platform-specific specifications in a later stage of the development lifecycle. The next subsection discusses the MDA in more detail.

1.1.1 Model Driven Architecture

The Model Driven Architecture is based on a pattern in which a Platform Independent Model (PIM) is transformed to a Platform Specific Model (PSM) (see also Fig. 1.1). The software is initially modelled in a PIM, which only

contains elements that are common to all targeted platforms. A PIM often uses high-level abstractions that are later refined to an implementation in the PSM. This PIM-to-PSM refinement takes the form of model transformation in the MDA. This transformation step can take additional input. Fig. 1.1 shows this extra input as a question mark to emphasise that no special requirements exist for this extra input. The MDA suggests that this extra input can be in the form of a Platform Model (PM). In the MDA, there can be multiple PIM-to-PSM transformation steps, as each PSM can serve as a PIM at a lower level of abstraction. The chain of PIM-to-PSM transformations usually terminates with a transformation to executable code.

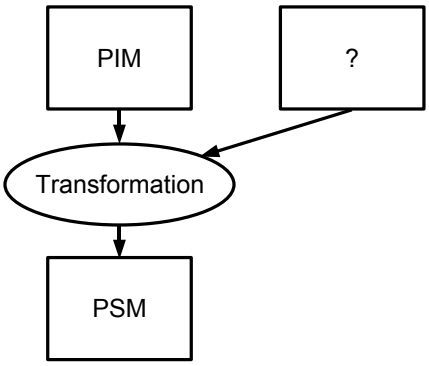

Figure 1.1: MDA pattern

Model transformations use *meta-models* as an additional *input*, as shown in Fig. 1.2. A meta-model describes the types of elements that can occur in a model through *meta-classes*. "Class", "Operation" and "UseCase", for example, are UML meta-classes. Instances of those meta-classes can occur in UML models. Meta-classes are similar to normal, object-oriented classes in that they can inherit from each other and they can have properties. A model *conforms to* a meta-model if it correctly uses the meta-classes described by that meta-model. The input meta-model in Fig. 1.2 provides a hierarchy of meta-classes that tells the model transformation when to trigger on a source element. The output meta-model provides the model transformation with the rules that the output model should follow.

In current MDA practise, most PIMs are transformed directly to PSMs, without using any extra input such as a platform model (see also Fig. 1.3). Tools like AndroMDA[4] and ArcStyler[5], for example, use the notion of *cartridges* to represent a particular PIM-to-PSM transformation. The model transformations implicitly assume a platform. This makes it much easier to write model transformations, since one only has to deal with the limited scope

[4] http://www.andromda.org/
[5] http://www.arcstyler.com/

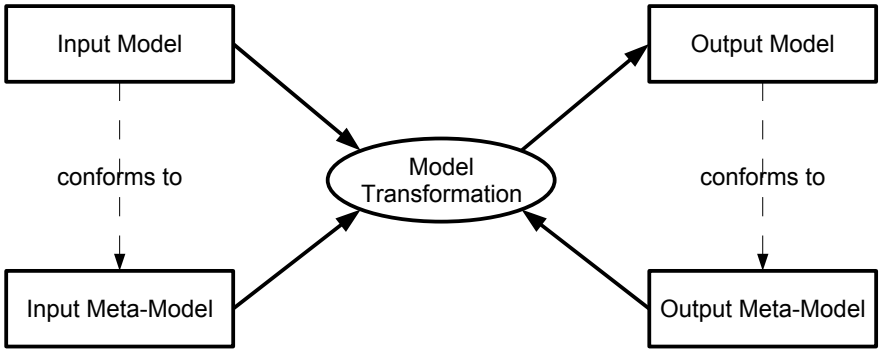

Figure 1.2: Model transformation pattern

of targeting a single, assumed platform. It is unclear, however, whether a model transformation can be used for other platforms than the one for which it was written. The only safe assumption is that each targeted platform requires its own dedicated set of model transformations.

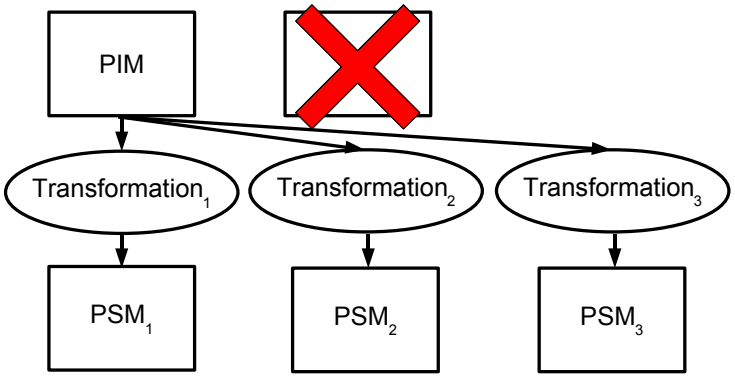

Figure 1.3: MDA practise

Another approach to MDA that applied often is shown in Fig. 1.4. In this approach, the PIM is absent and one PSM serves as the input for transformations that generate other PSMs. This PSM-to-PSM approach occurs mostly in situations where one has to translate from one language/technology to another, such as translating from Rational Rose to UML2 Tools and from UML to OWL[6], but also to translate from one Java API to another [CDZ04].

In reality, this means that only a relatively small number of "general" platforms can be targeted, such as Java. As has been explained before, Java is actually a family of platforms, ranging from J2ME to J2EE. On top of this generalisation, specific platform assumptions are often made. In the case of the

[6] http://www.eclipse.org/m2m/atl/usecases/

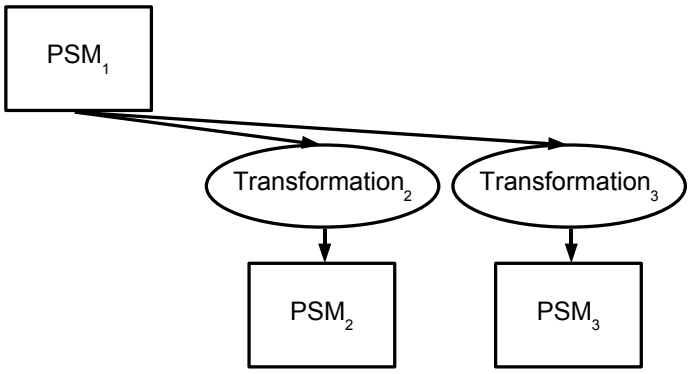

Figure 1.4: Alternative MDA practise

Java platform family, there is not enough commonality between the different Java versions to come up with a complete platform to build on. For example, the API for graphical user interfaces (GUIs) is radically different on the mobile phone J2ME MIDP than it is on the standard J2SE; there is no common Java API for GUIs. For the purpose of targeting a "complete" platform, an assumption for a specific GUI library is made, thereby reducing the number of supported Java platforms. This way of "streamlining" the PIM-to-PSM transformations to a single, assumed Java platform is used a lot to *escape* the problem of maintaining model transformations for multiple platforms.

When we look at the nature of PIM-to-PSM transformations directly, it becomes clear that the single Java platform approach is a very bad solution indeed. Most of these transformations consist of several smaller refinement steps that, when considered individually, are reusable over multiple platforms. It is only the *combination* of refinement steps that limits their applicability to one specific platform. For example, one refinement step could target all Java 2 platforms by transforming the types of UML Properties [OMG05c] with a multiplicity greater than one to the Java 2 Collections framework. The Java 2 Collections framework is included in several Java platforms, including J2ME Personal Profile and J2SE. If this refinement step is applied in combination with a refinement step that targets the Java Swing graphical user interface framework, the target platform is already limited to J2SE. In order to reuse individual refinement steps that occur in a transformation, each refinement step must be modularised in its own model transformation. One can then reuse those refinement transformations in different transformation configurations, as is done in Stepwise Refinement [BSR04].

If the PIM-to-PSM transformations are split up in multiple, step-wise refinement transformations, then the bulk of the maintenance problem moves to the transformation configuration. Software developers must keep track of which refinement transformations must be applied to target a specific plat-

form, and in which order they must be applied. An increasing number of MDA practitioners apply *workflow scripts* to specify which PIM-to-PSM refinement transformations they want to apply and in which order[7][8]. Such a workflow script can take the form of an Ant build script[9], which is a Makefile equivalent for Java.

The MDA picture then changes to what is shown in Fig. 1.5. Each configuration of refinement transformations is represented by a separate build script. The execution of the build scripts results in a separate PSM for each of those build scripts. Refinement transformations can now be reused in multiple configurations, targeting multiple platforms. While these separate refinement transformations lay the basis for lifting the platform maintenance burden, they do not solve our initial problem. It is still unclear whether a refinement transformation can be used for other platforms than the one for which it was written, since all platform dependencies are still implicit.

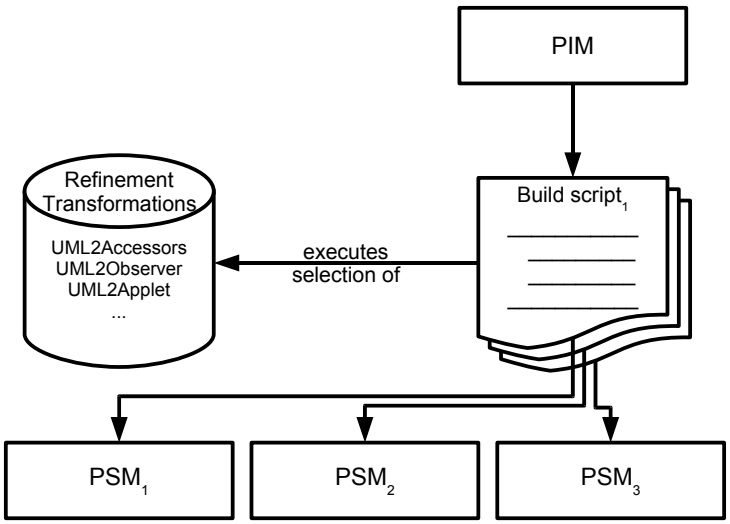

Figure 1.5: MDA improved

As the software developer puts the build script together that executes the refinement transformations, most of the effort goes into checking that (1) the refinement transformations work together and that (2) they are executed in the right order [MTR05]. It is an extra burden to also (3) consider the platform dependencies that each refinement transformation introduces. One approach is to test the generated software on the target platform to tell if the software works on that platform. Testing on each platform is a time-consuming activity, however, and may even leave certain incompatibilities undetected until after

[7] http://www.openarchitectureware.org/
[8] http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)
[9] http://wiki.eclipse.org/index.php/AM3_Ant_Tasks

deployment. Another approach is to use an automated configuration process that enforces the satisfaction of constraints, including platform dependency constraints. Such a configuration process does not exist for the MDA today.

1.2 Research Objective

The problem of platform diversity is a very broad one. Solutions to the problem range from standard libraries that offer a unified API, such as the C standard library[10], to virtual machines that offer a unified binary format, such as the Java virtual machine. As we have previously illustrated, these abstraction layer solutions often fall short of solving the platform diversity problem.

The MDA offers solutions for dealing with platform diversity. It achieves this by abstracting from specific platforms in the PIM, followed by – alternative – transformations that refine the PIM into a PSM. The MDA suggests that multiple, step-wise refinement transformations are used to generate a PSM. These refinement transformations are often reusable over multiple platforms.

We don't know, however, when we can – and when we cannot – reuse a PIM-to-PSM refinement transformation for another platform than the one for which it was written. Each refinement transformation typically introduces certain platform dependencies in the PSM, but these platform dependencies remain implicit to this day (see Fig. 1.6). It is our objective to make these platform dependencies explicit, so that we may reason about them. We plan to achieve this objective in several stages:

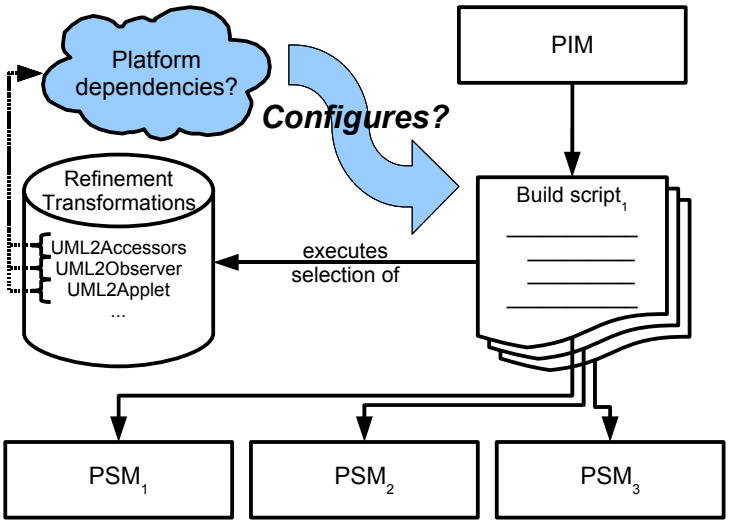

Figure 1.6: Integrating platform dependencies

[10] http://en.wikipedia.org/wiki/C_standard_library

- Build a body of explicit domain knowledge that describes what a platform is and extend this domain knowledge for the family of Java platforms.
- Develop a method to describe platform instances based on this platform domain knowledge.
- Develop a method to describe platform dependencies based on this platform domain knowledge. Descriptions of platform dependencies must remain valid even if the targeted platforms evolve.
- Develop a method to check whether platform dependencies are satisfied by platform instances.

Explicit platform dependencies are only useful if they can be enforced during the configuration of PIM-to-PSM refinement transformations, as shown in Fig. 1.6. As the MDA offers no process for the configuration of step-wise PIM-to-PSM refinement transformations, our second objective is to introduce a configuration process with automated support for platform dependencies. This must result in a *framework for managing platform dependencies* for the MDA that:

- improves the maintainability of a PIM-to-PSM transformation configuration by enabling safe reuse of individual refinement transformations in such a configuration,
- assists in finding the most appropriate PIM-to-PSM transformation for a specific platform from a number of alternatives,
- integrates with existing software development technologies, in particular the software development technologies that target Java platforms.

The thesis statement of this dissertation can be summarised as:

> *In order to deal with platform diversity, we believe that* ***(1) platform domain knowledge must be made explicit****. By making this knowledge explicit, we can* ***(2) reason about the extent of platform dependencies with regard to platform instances*** *as well as* ***(3) compare appropriateness of alternative refinement transformations*** *and* ***(4) enforce a safe configuration of refinement transformations via their introduced platform dependencies****.*

1.3 Approach

1.3.1 Explicit Platform Models

We propose to use an explicit platform model, which serves as an ontology for describing platforms. This ontology is used as a basis to express platform instances as well as platform dependencies. By explicitly specifying the platform dependencies for each reusable model transformation, each transformation can be guaranteed as valid for a well-defined class of platforms. Because platform instances use the same platform model as an ontology, the platform model enables us to determine which platforms satisfy which platform dependencies. It is expressed in the Web Ontology Language (OWL) [SWM04], which is an extensible language for describing ontologies. Ontologies are commonly used to represent domain knowledge and to provide a community of users with a controlled vocabulary. We use the OWL DL variant, which corresponds to description logic (DL) [BCM+03] and allows us to apply automatic reasoning. We represent platform dependencies in OWL as classes. The representation of a platform dependency is called a *platform dependency constraint*. Consider, for example, the "JavaAWTPlatform" platform dependency constraint shown in Fig. 1.7.

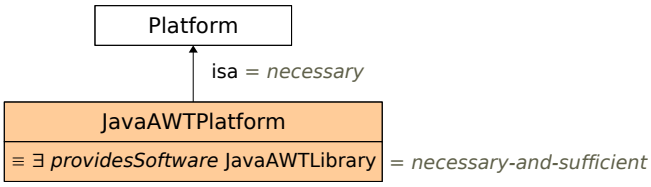

Figure 1.7: Example platform dependency

"JavaAWTPlatform" is represented as an OWL class with a *necessary* constraint as well as a *necessary-and-sufficient* constraint. Whereas it is *necessary* that each "JavaAWTPlatform" is a "Platform", being a "Platform" is not *sufficient* for also being a "JavaAWTPlatform". Providing the "JavaAWTLibrary" software, however, is *necessary-and-sufficient* for being a "JavaAWTPlatform". The "JavaAWTPlatform" constraint can be checked against OWL instances that represent platform instances. Each OWL instance – or *individual* – that satisfies the conditions of the "JavaAWTPlatform" class can be considered as an instance of that class. Each platform instance representation that is an instance of a platform dependency constraint, satisfies that platform dependency constraint.

Our platform model is not monolithic, but is divided into a hierarchy of modules. Fig. 1.8 shows how the platform model is organised. The central part of the platform model is made up of several "vocabulary ontologies",

where the word "vocabulary" refers to the fact that the domain concepts are all introduced in these ontologies. The main Platform vocabulary ontology describes the general concept of Platform and its parts. The Java vocabulary ontology extends the Platform ontology for the domain of Java platforms. The Java platform ontology is in turn extended by several vocabulary ontologies that describe concrete Java variants: JDK 1.1, J2SE 1.2, J2ME PP 1.0, etc. Platform dependency constraints are expressed in terms of these vocabulary ontologies, while they are stored in a separate OWL ontology. This separate OWL ontology is not considered to be a vocabulary, since it expresses only platform dependency constraints rather than platform domain concepts.

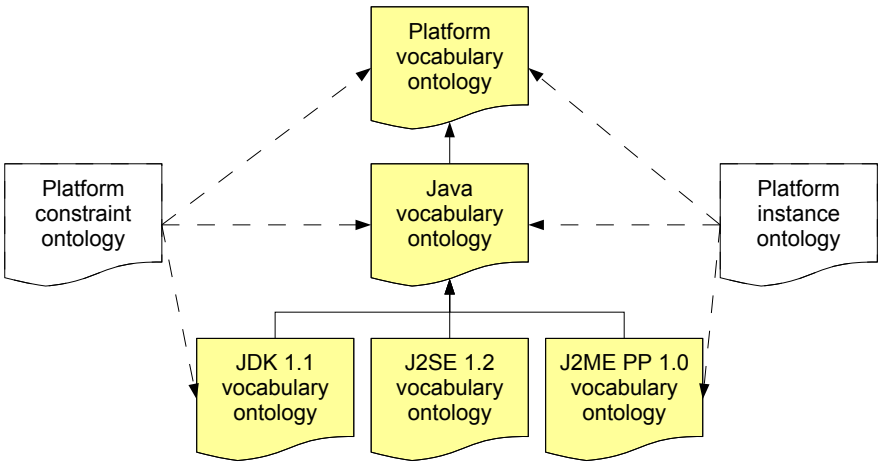

Figure 1.8: Platform model overview

Platform instances are also modelled as a separate OWL ontology and refer to the same platform vocabulary ontologies as the platform dependency constraints. An automatic DL reasoner, such as Racer [MH03], can be used to verify whether a platform instance satisfies the platform dependency constraints of a model transformation. In addition, it can determine which platform dependency constraint is *most specific* and hence forms the closest match to a platform instance. This allows for optimisation towards a platform instance by automatically selecting the optimal transformations for a given platform.

The explicit platform dependency constraints for each model transformation also make it possible to automatically derive the overall platform constraints for a configuration of transformations. This allows us to verify which build script (see Fig. 1.5) is *most-specific* and valid for a given platform. What is left is the integration of our platform model with a configuration approach for the MDA. That way, we can leverage the platform dependency information during the configuration process.

1.3.2 Platform-Driven Configuration

Configuration is a complex topic that has already been widely researched. Within the field of software engineering, most research on configuration has been conducted by the Software Product Line (SPL) [CN01] community. SPLs are concerned with leveraging the commonalities between related software products. They integrate a number of software-intensive products that share a significant amount of functionality. As such, any software that is developed using the MDA approach can be considered as an SPL, since each platform-specific software product shares significant functionality with other platform-specific versions of that software product. This shared functionality is typically specified in the PIM. In [Bos06], Bosch also argues that SPLs have been especially successful in the area of mobile and embedded devices, in which platform diversity is the rule rather than the exception. This relationship between the MDA and SPLs allows us to build on the configuration technology used in SPLs for the configuration of our PIM-to-PSM refinement transformations. Whenever the MDA and SPL technology are used in combination, we will speak of *MDA-based SPLs*.

In contemporary SPL practise, the principles of SPLs have been extrapolated to shift the bulk of the engineering effort to the product line's reusable assets [Kru06]. The effort for deriving software products from these assets is reduced to creating a Product Model that contains all configuration information for that product. The product line's infrastructure includes a generator ("SPL Configurator") that generates the product's implementation from the Product Model, as shown in Fig. 1.9a. In Generative Programming [CE00], Product Models are often expressed in terms of a Domain-Specific Language (DSL) – or a Domain-Specific Modelling Language (DSML) [LBM+01][TR03]. DSMLs are a specific kind of DSL that use meta-models as a language definition formalism. We have explained previously how PIMs and PSMs in the MDA also have meta-models, which are used by the PIM-to-PSM refinement transformations.

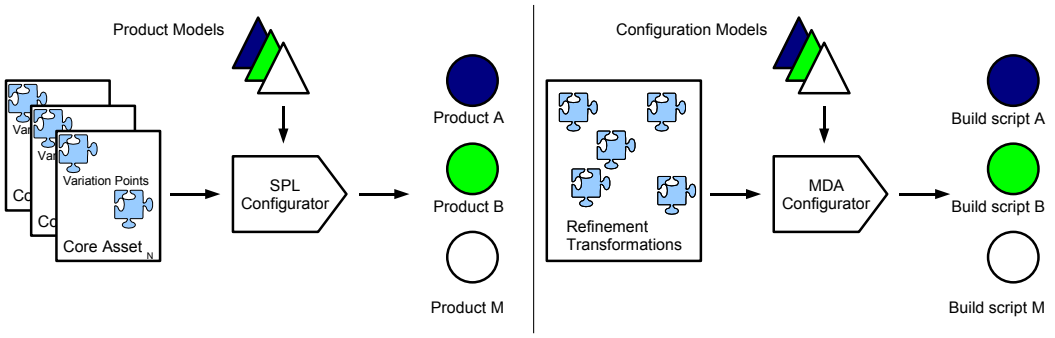

Figure 1.9: (a) Software product line configurator and (b) MDA configurator

In the MDA world, the DSML for Product Models can take the form of a *configuration language* that can express configurations of PIM-to-PSM refinement transformations. The "SPL Configurator" part of the SPL's infrastructure (see Fig. 1.9a) is normally used to generate the SPL's products, but it can also be used to generate the build script that was shown in Fig. 1.6. The SPL pattern then changes to what is shown in Fig. 1.9b. A number of refinement transformations are selected in a configuration model. The "MDA Configurator" then takes these configuration models as input and generates build scripts that invoke the selected refinement transformations.

The "MDA Configurator" itself can be implemented as a model transformation, as shown in Fig. 1.10, provided that we have a meta-model for our configuration language as well as our build script language. We can even use the same model transformation technology for PIM-to-PSM refinement transformations and the "configuration-to-build-script" transformation.

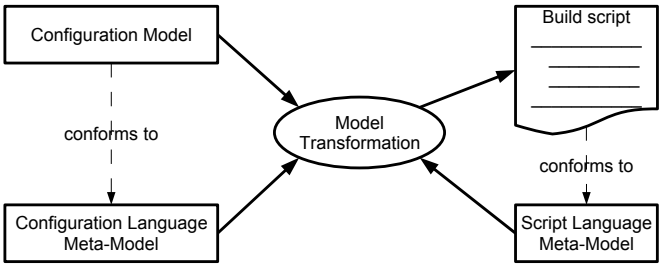

Figure 1.10: Software product line generator transformation

In order to integrate our platform models into a software development process based on MDA and SPL, we propose to represent all product configuration rules – including alternative model transformations – inside a configuration language meta-model. Several meta-classes in this meta-model are then annotated with references to platform dependency constraints. The annotated configuration language meta-model can be used to determine the platform dependency constraints of each configuration model.

1.4 Contributions

- **A common platform domain model**
  We present how OWL-DL can be used to define a general platform model that serves as a common ontology for specific platform sub-domains. This work has been presented in [PVW+04], [Wag05] and [WJ05]. We also present how the general platform model can be extended for the platform sub-domain of Java Runtime Environments (JREs). This work has been presented in [Wag05], [WJ05] and [WV07].

- **A method for describing platform dependencies and platform instances**
  We present how platform dependency constraints and platform instances can be described, based on the common platform model and extensions. The platform dependencies can be compared against platform instances to check if they are satisfied. In addition, we can determine which platform dependencies are *more specific* than other platform dependencies and form a closer match to the targeted platform. This work has been presented in [Wag05], [WJ05] and [WV07].

- **A framework for platform dependency management**
  We present how the platform model can be integrated in a software development process based on the MDA and SPLs. The relationship between the MDA and SPLs is explored as part of this work and a common configuration approach based on DSMLs is used. This work has been presented in [WV06] and [WV07].

- **A framework for platform-driven optimisation**
  We present how platform dependencies can be used as the basis for selecting optimal model transformations (or SPL features). In addition, we present how this can be extrapolated for the selection of optimal configurations. This work has been described in [Wag05], [WJ05] and [WV07].

- **A case study that applies the explicit platform model in an MDA/SPL setting**
  We have developed a non-trivial case study of a cross-platform Instant Messaging client that demonstrates the merits and limitations of our approach. The case study is available at http://ssel.vub.ac.be/ssel/research:mdd:casestudies and is used in [Wag05], [WJ05] and [WV07].

- **Tool support**
  We have developed a tool, named *PlatformKit*, that implements platform dependency management and platform-driven optimisation based on the Eclipse Modeling Framework (EMF) [BSM+03]. The PlatformKit tool is available at http://ssel.vub.ac.be/ssel/research:mdd:platformkit and is described in [WV07]. In addition, we have developed the *Jar2UML* tool that reverse engineers Java class libaries to UML models. Those UML models are used by PlatformKit to determine compatibility between different Java platforms. Jar2UML is available at http://ssel.vub.ac.be/ssel/research:mdd:jar2uml. Finally, we have added several improvements to the ATLAS Transformation Language tool for the purpose of our case study. Those improvements have been integrated back into the main ATL code, which is available at http://www.eclipse.org/m2m/atl. One of the ATL improvements, module superimposition, is described in [Wag08].

1.5 Dissertation structure

Fig. 1.11 shows an overview of the structure of this dissertation. Following this chapter, a background chapter on the Model-Driven Architecture and a background chapter on ontologies provide the basis for our approach to platform modelling. The next chapter discusses our platform modelling approach itself. This is followed by a background chapter on Software Product Lines that provides the basis for our MDA configuration approach. Our MDA configuration approach itself is discussed in the subsequent chapter. The next chapter discusses our tool support and the final chapter discusses the conclusions of this dissertation. We will now give a detailed description of each chapter.

Chapter 2: Model-Driven Architecture This chapter gives an in-depth explanation of the MDA. The main concepts of the MDA, such as model,

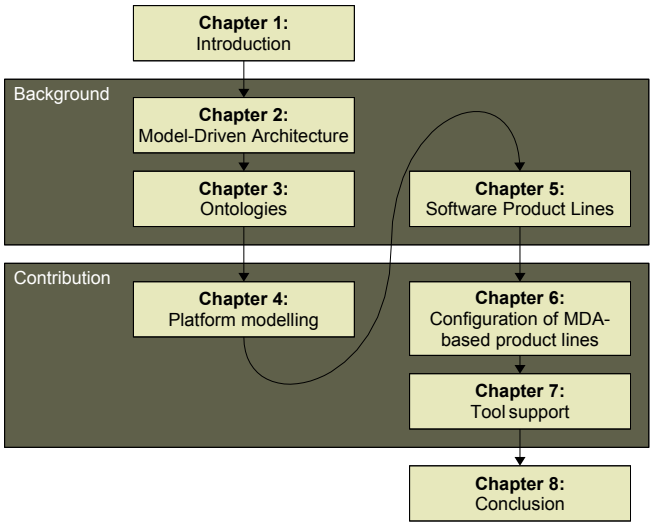

Figure 1.11: Dissertation structure overview

meta-model and model transformation, are discussed in detail. The different roles that models play within the MDA framework are discussed: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and Platform Model (PM). Meta-models are discussed as a technical grounding for the definition of modelling languages. The Meta Object Facility (MOF), which is a standard meta-modelling language, is discussed in particular. In addition, the Eclipse Modeling Framework is discussed, which is a meta-modelling framework that is derived from MOF. The special position of UML stereotypes within the MDA is also discussed, as they are partly situated at the meta-model level. Finally, model transformations are discussed. The standard MOF Query/View/Transformation (QVT) language is discussed, as well as the ATLAS Transformation Language (ATL), which is a QVT predecessor. Examples are given of PIM-to-PSM transformations written in ATL.

Chapter 3: Ontologies In this chapter, we explain what ontologies are. The OWL ontology language is discussed in detail, as it forms the technical basis for our platform model. Each relevant OWL language construct is explained in detail. The consequences of using a particular OWL construct for logic inference are discussed. Throughout the chapter, several pitfalls of using OWL are identified and their theoretical basis is given. Together with the previous chapter, this chapter forms the background for the first half of this dissertation.

Chapter 4: Platform modelling The previous two chapters have discussed

sufficient background information to explain the workings and use of our platform model. This chapter starts with a motivation of the need for explicit platform models. Platform ontologies are presented as a solution for safe PIM-to-PSM transformation reuse as well as a solution for decoupling platform dependency constraints from platform instance descriptions. A common vocabulary for describing platforms is then introduced, as well as an extension of this common vocabulary for the domain of Java platforms. As the domain of Java platforms must be described in a rather large and detailed ontology to be of any practical use, an approach is presented for automatic generation of such ontologies. The automatic generation of ontologies ensures that the platform vocabulary can be kept up to date with new Java platforms. The chapter then goes on to discuss platform instance specifications as well as platform dependency constraints. The platform dependency constraints can be classified in a hierarchy and constraint satisfaction can be checked against platform instances. Finally, the limitations of our platform modelling approach are discussed. This chapter also concludes the first half of this dissertation.

Chapter 5: Software Product Lines The second half of this dissertation starts with an in-depth explanation of SPLs – a mode of software development that includes the MDA. The different stages in SPL development are discussed: Commonality and Variability Analysis (CVA), Feature Modelling (FM) and Configuration. As SPL Configuration is the most interesting part for our purposes, this topic is discussed in most detail. A configuration approach using Domain-Specific Modelling Languages (DSMLs) is presented in particular, as it connects best to MDA-based product lines.

Chapter 6: Configuration of MDA-based product lines The previous chapter lays the groundwork for this chapter, in which an integrated configuration approach for MDA-based SPLs is presented. This chapter starts with a discussion of interaction problems between PIM-to-PSM transformations. The similarity with feature interactions in SPLs is shown and justifies the application of SPL solutions to the MDA. The difference between interaction constraints and platform dependency constraints is also discussed, showing the need to integrate platform dependency constraints. An integration of platform constraints with domain-specific configuration languages is presented. We then show how this integration can be done in a loosely coupled way by using an intermediate "shadow" model. This "shadow" model can be used in two different scenarios and allows the processes of these scenarios to remain very similar (reuse of sub-processes). Finally, the limitations of our configuration

approach are discussed.

Chapter 7: Tool support This chapter adds practise to our theory by providing an in-depth discussion of our PlatformKit tool and how it can be used to support several platform modelling and configuration scenarios. First, the architecture of PlatformKit is explained. All components on top of which PlatformKit is built are discussed here. Following that, a number of tasks that are supported by PlatformKit are discussed. These tasks range from setting up an MDA-based SPL to platform-driven deployment of software products. Finally, the limitations and future directions of PlatformKit are discussed.

Chapter 8: Conclusion This chapter concludes this dissertation with an evaluation of the approach we have presented. An overview of our contributions is given and future research directions are explored.

Chapter 2

Model-Driven Architecture

2.1 Introduction

The Model-Driven Architecture (MDA) was introduced by the Object Management Group (OMG) as an approach to use models in software development. Its goals are portability, interoperability and reusability through architectural separation of concerns. MDA intends to provide an approach for:

- "specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and
- transforming the system specification into one for a particular platform." [MM03]

The terms *platform* and *platform independence* are used extensively in MDA. The MDA Guide defines a platform as follows:

> "A *platform* is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented." [MM03]

This definition introduces the notion of interfaces and usage patterns. It also positions a platform as an abstraction layer that can be used without knowing how platform functionality is provided. We have previously defined

a platform as *the hardware and software combination on top of which our software runs*, which is a broader definition of platform.

Platform independence refers to the quality of being independent of the features of a platform and is a matter of degree. A common technique for achieving platform independence is to target a technology-neutral virtual machine. A virtual machine can also be considered a platform. Any model targeting that virtual machine is considered specific to that platform, but independent of any underlying platform. When specifying a system in a platform-independent way, one already needs to have an idea of which platforms are targeted [TBA04]. The system's specification is independent of all these platforms if it only depends on features that are *common* for all platforms – or can be *mapped* to all platforms.

MDA contains the word *architecture*, which is an overloaded term in software engineering. The MDA guide uses the following definition to relate the term *architecture* to its meaning in MDA:

> "The *architecture* of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors. The Model-Driven Architecture prescribes certain kinds of models to be used, how those models may be prepared and the relationships of the different kinds of models." [MM03]

The Model-Driven Architecture refers to the architecture of the software development process, summarised by the MDA pattern. In Chapter 1, we have already introduced the basic MDA pattern: to use Model Transformation to map a Platform-Independent Model (PIM) to a Platform-Specific Model (PSM). Fig. 2.1 shows the envisioned impact of this pattern: the left side shows the traditional lifecycle of iterative development. In theory, the feedback loop should go all the way up to the requirements level, keeping all software artifacts synchronised.

In practise, the fact that process stage transitions are only automated from the code level downward makes that it is not feasible to feed changes back to the top level. The feedback effort is normally rewarded by an updated system that incorporates the feedback. That updated system is our real goal, not feeding back changes to higher abstraction levels.

If we provide feedback to the top level, this will not result in an updated system. In fact, we have to manually translate the feedback to the lower levels of abstraction until we reach code level. Only at code level, automation picks up and the feedback loop is closed. That's why currently feedback is mostly provided directly at code level and all software artifacts at a higher level gradually become outdated.

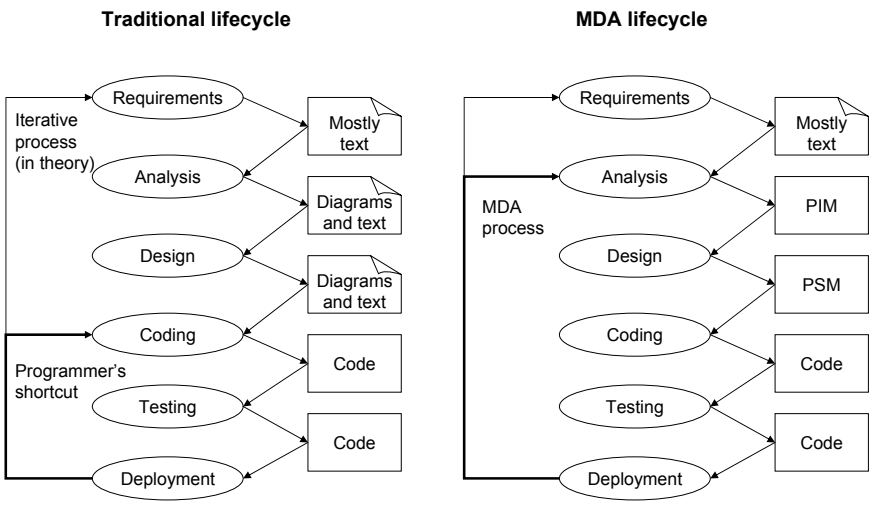

Figure 2.1: Impact of MDA on the development process (source: [KWB03]).

The right side of Fig. 2.1 shows how the MDA intends to mitigate this problem. The analysis and design artifacts are made part of the automation pipeline. It then becomes easier to lift the feedback loop of iterative development up to the analysis level, where automation picks up and closes the feedback loop.

The core concepts in any MDA-based process are:

- Models
- Meta-models
- Model Transformation

The following sections discuss these core concepts, as well as the relationship of MDA to existing research.

2.2 Models

MDA is centred around *models* and how those models may be used together to create the resulting software system. The MDA guide defines a model as follows:

> "A *model* of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language." [MM03]

This is a very broad definition of a model, which unfortunately mixes up the structure of a model with its presentation. The *structure* of a model is defined by the *abstract syntax* of the language that a model is written in. The *presentation* of a model is defined by the *concrete syntax* of the modelling language. Even though the definition covers the presentation of models, the focus lies on the structure of a model in the MDA, as is made clear in section 2.3. Models can play a number of particular roles in an MDA-based process. Four kinds of models with a particular role are distinguished in the MDA:

- Computation Independent Model (CIM)
- Platform Independent Model (PIM)
- Platform Specific Model (PSM)
- Platform Model (PM)

The standard language for expressing these models is the Unified Modeling Language (UML) [OMG05c]. The following sections discuss each of the listed models.

2.2.1 Computation Independent Models

The Computation Independent Model focuses on the requirements for the system and the environment of the system. It is sometimes called a *domain model* and serves as a vocabulary that is familiar to the practitioners of the system's domain. A CIM is more than a domain model, however, since it also expresses the system requirements using the domain concepts as a vocabulary. A CIM does not show details of the structure or processing of the system. The CIM is intended to help bridge the gap between domain experts and system design experts.

It is difficult to say exactly when a model is still "computation independent". It may show the structure of the domain, but not the details of that structure. It also may show the behaviour of the domain, but not any processing details specific to the system we aim to specify. Let's consider an example system of an instant messaging client that shows a list of people who are online and that can send messages to those people. Fig. 2.2 shows what such an instant messaging client can look like.

A CIM of such an instant messaging client describes the domain concepts for instant messaging, such as "contact", "contact list" and "message". For each of these concepts, a number of behavioural and structural features (operations and attributes) can be described. The CIM does not describe what steps the system must perform to achieve a certain behaviour. Fig. 2.3 shows a UML Class diagram of the instant messaging client domain concepts. Note

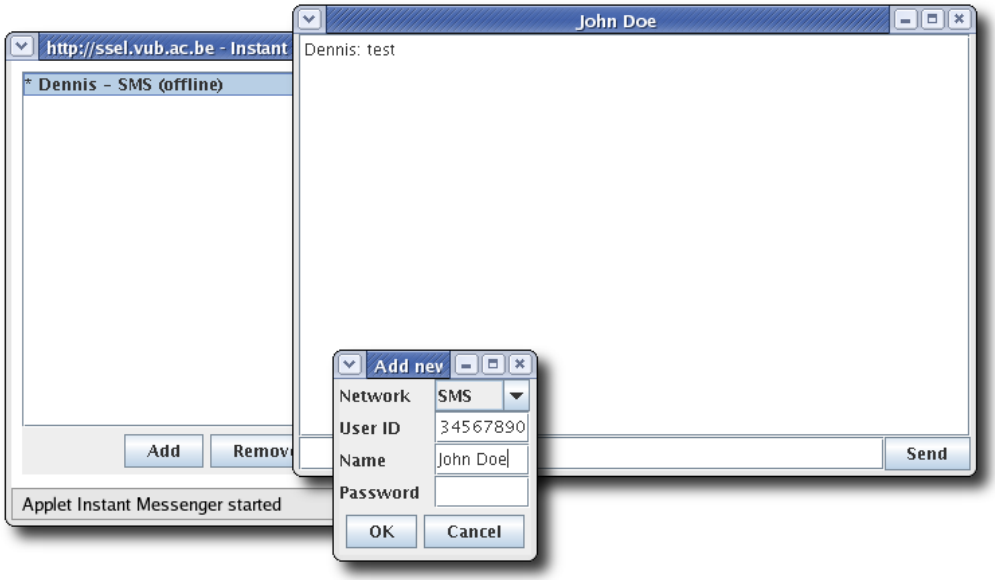

Figure 2.2: A screenshot of an instant messaging client running on a PC.

that the example CIM is not a "pure" CIM: it contains "computation dependent" stereotype applications, such as `Observer` and `Observable`. The cause of this is that a CIM is typically refined manually into a model that includes computation details. The original CIM no longer exists in the end and is actually a part of the Platform Independent Model. Since the CIM typically models the domain concepts up to the extent that the intended system requires, it is often not directly applicable outside the boundaries of the intended system.

2.2.2 Platform Independent Models

A Platform Independent Model exhibits a specified degree of platform independence, such that it is suitable for a number of similar platforms. This implies that the targeted platforms must be known beforehand to a certain degree. It also implies that a PIM is still specific to the chosen group of platforms.

In the case of our instant messaging client example, the PIM is meant to target Java-based client platforms. It is independent of any specific Java client platform, such as J2SE 1.4 or J2ME Personal Profile 1.0, but still depends on Java in general (language, object model, execution model, common API, etc.). Fig. 2.4 shows a UML Class diagram of the global instant messaging client architecture. While the CIM is only concerned with instant messaging client domain concepts, the PIM also includes computational information, such as the fact that an InstantMessagingClient is an `<<Applet>>` that subscribes to all available Networks as an `<<Observer>>`. This PIM also introduces Java-specific elements: InstantMessagingClient implements the Excep-

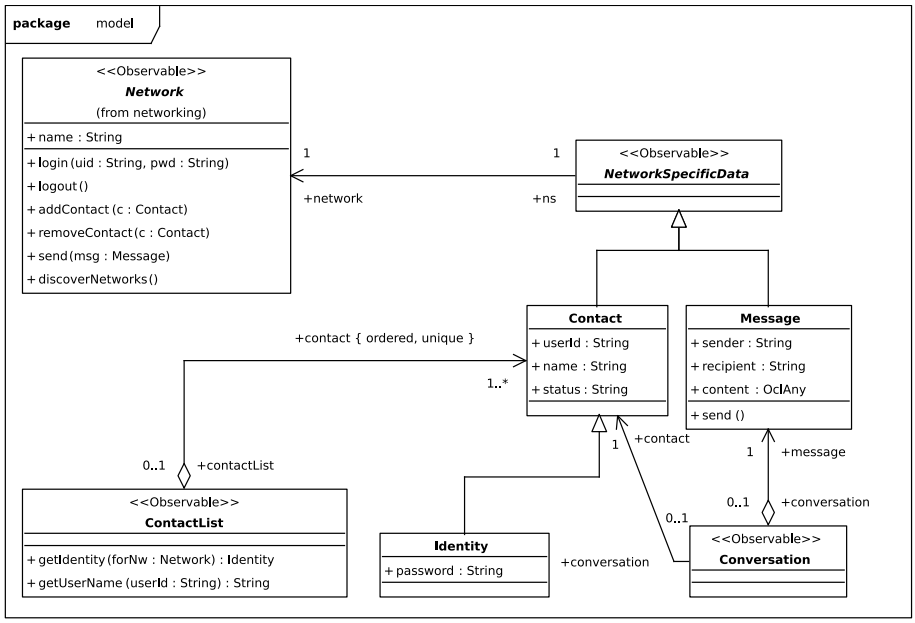

Figure 2.3: A UML Class diagram of the instant messaging client CIM.

tionReporter interface, which includes a reference to `java.lang.Exception`. Since `java.lang.Exception` exists in all our targeted Java platforms, it is considered platform-independent for our purpose.

2.2.3 Platform Specific Models

A Platform Specific Model combines the specifications in the PIM with the details that specify how the system uses a particular (kind of) platform. This means that it introduces platform dependencies. The term PSM is relative and only means something when compared to a PIM. There can be multiple levels of PSMs. Consider a PSM that is specific to a (narrowed down) group of platforms. This PSM can be considered a PIM when compared to a PSM that is specific to one concrete platform out of that group.

Fig. 2.5 shows the fully refined platform-specific version of the PIM Class diagram shown in Fig. 2.4. Fully refined means that this is the lowest level PSM, just before Java code is generated:

- UML properties that have a multiplicity higher than 1 have been transformed to use Java collection types: InstantMessagingClient::network has become a `java.util.List`, while InstantMessagingClient::conversation has become a `java.util.Set`.
- Accessor operations have also been generated for each public property (get..., set..., add..., etc.), while those public properties have been made private.

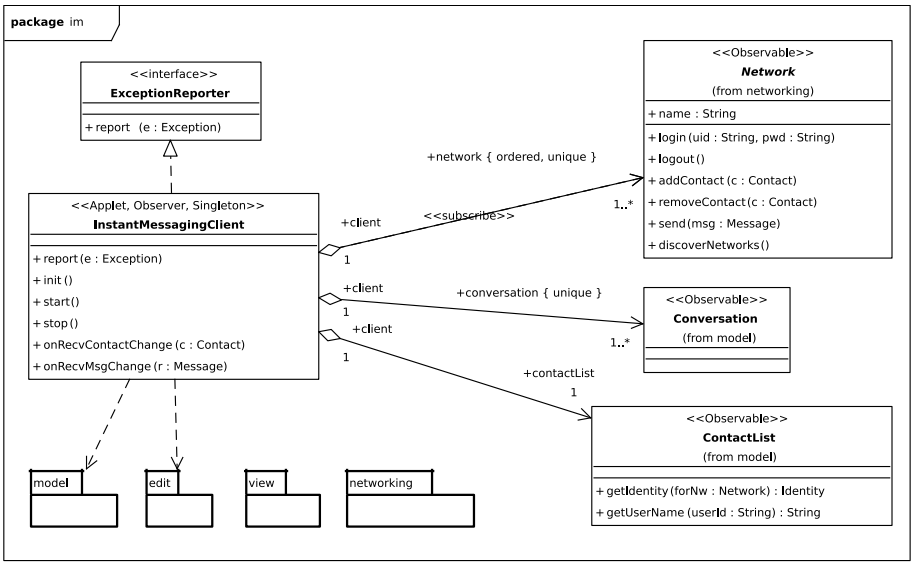

Figure 2.4: A UML Class diagram showing part of the instant messaging client PIM.

- The observer pattern has been implemented using the `java.util.Observer` interface and the `java.util.Observable` class.
- All `<<Applet>>` classes have been implemented as `java.applet.Applet` subclasses.
- All `<<Singleton>>` classes now include the singleton infrastructure (a static 'instance' property and 'getInstance()' operation).

2.2.4 Platform Models

A Platform Model provides the technical concepts that represent the parts that make up a platform. It also provides concepts for specifying the use of a platform by a software system, which can be used in a PSM. As an example, the CORBA Component Model (CCM) [OMG06a] provides concepts as EntityComponent, SessionComponent, ProcessComponent, Facet, Receptacle, EventSource, etc. A software system can use these concepts to specify how it uses the CCM.

A Platform Model also specifies requirements on how a software system is connected to the platform. As an example, the UML profile for CORBA [OMG02] provides a language to use when modelling CORBA systems. The rules of this language represent the requirements of the connection to the platform.

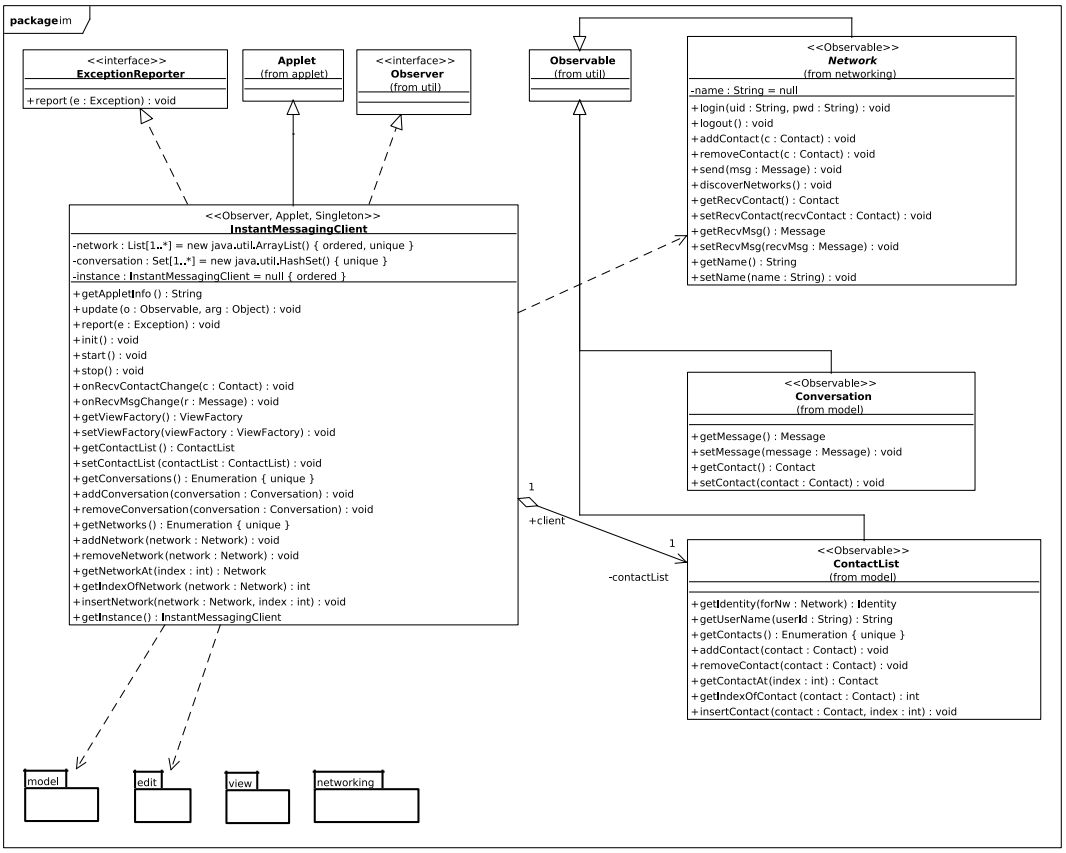

Figure 2.5: A UML Class diagram showing part of the instant messaging client PSM.

This notion of Platform Model is mostly useful to model *abstract platforms* [ADvSP04], which represent a common interface to all targeted platforms. In the instant messaging client example, this Platform Model consists of a simplified representation of some of the Java API that is common to all targeted Java platforms (e.g. `java.lang.Exception`) and several UML profiles (Applet, Singleton, Observer, ...) that abstract from the way a specific Java platforms implements certain functionality. A Platform Model can also consist of the API for a specific Java platform, such as J2SE 1.4. The PSM typically refers to such a Platform Model. Note that the model transformations that map a PIM to a PSM introduce the references to such a Platform Model.

Our notion of Platform Model goes further than that: a Platform Model is meant to *drive* the mapping from PIM to (lowest level) PSM. This may include selecting all the appropriate model transformations for a complete mapping to the targeted platform. This notion of Platform Model is discussed later in this dissertation.

2.3 Meta-models

While the MDA leaves open the specific technical grounding for the definition of models, the standard way to define the language of the models, or abstract syntax to be more precise, is by using a *meta-model*. Standard practise is therefore that each model has a meta-model. The term *meta* is used to say something about something else. Hence, a meta-model can be defined as a model of a model. Since the term *meta* is a relative term, the OMG has introduced a standard, 4-level meta-modelling framework that disambiguates the term *meta*. This framework is shown in Fig. 2.6.

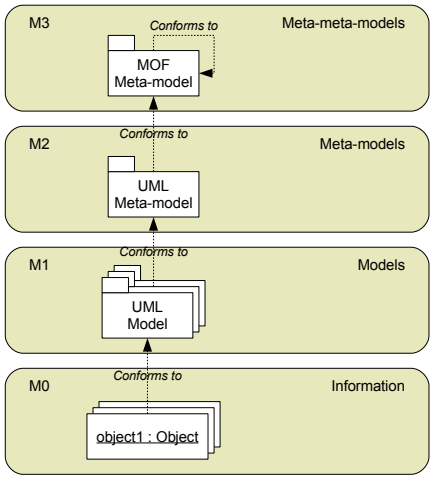

Figure 2.6: The 4-level OMG meta-modelling framework.

The OMG standard for the definition of meta-models is the Meta Object Facility (MOF) [OMG06b]. A framework that aims to follow the MOF standard is the Eclipse Modeling Framework (EMF) [BSM+03]. Both MOF and EMF are explained in the following subsections.

2.3.1 Meta Object Facility

The Meta Object Facility provides a framework for describing *meta-data*, which is defined as "data about data" [OMG06b]. At the heart of MOF lies the MOF Model, which serves as the standard meta-modelling language for the MDA. The MOF Model comes packaged in two versions:

- **Essential MOF (EMOF)**
  A subset of the MOF Model that forms a kernel meta-modelling language. EMOF supports basic meta-class entities, such as Class, Property, Operation, Package, DataType, PrimitiveType and Enumeration.

- **Complete MOF (CMOF)**
  Represents the entire MOF Model, which includes EMOF. CMOF does not introduce any new meta-classes, but extends existing meta-classes instead.

This section focuses on the EMOF subset of the MOF Model. Fig. 2.7 shows the part of the EMOF meta-model that models MOF Classes. Note that this meta-model is situated on level M3 (see Fig. 2.6). MOF Classes are used to represent the meta-classes in a meta-model (M2).

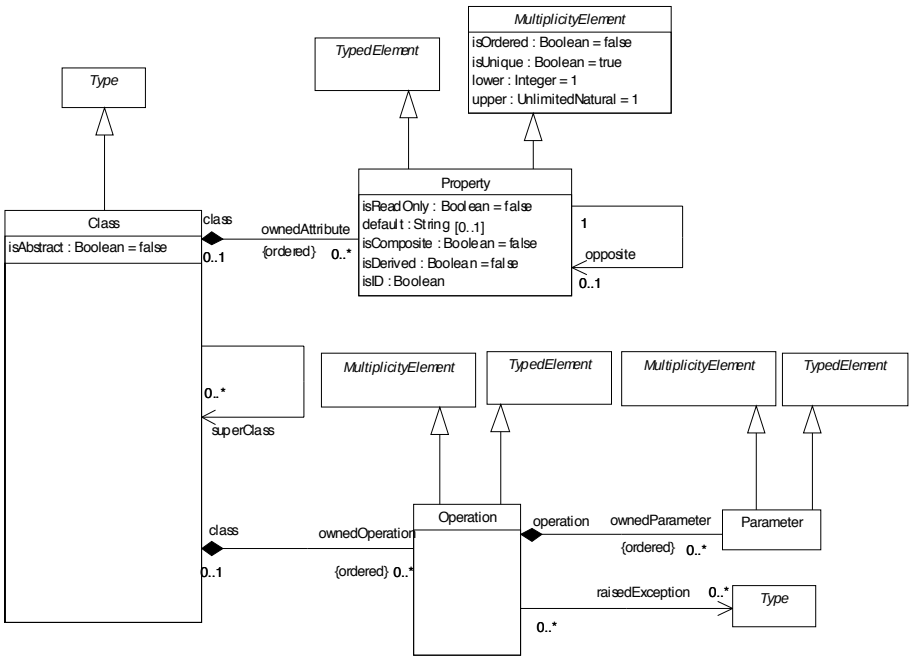

Figure 2.7: EMOF meta-model part for MOF Classes (source: [OMG06b]).

The UML meta-model, for instance, is expressed in this way. Fig. 2.8 shows the root diagram of the UML meta-model. The meta-classes shown in this diagram are all instances of MOF Class and their properties are instances of MOF Property. This particular diagram describes the properties that all

UML Elements have in common: Elements can own Elements and they can be annotated by Comments.

In addition to the standard semantics of classes, properties and associations, the UML meta-model also makes extensive use of the property *subsets* and *union* construct. The ends of the association between Element and Comment, for example, *subsets* the ends of the association between Element and Element; "owningElement" subsets "owner" and "ownedComment" subsets "ownedElement". This means that all values of "owningElement" are also contained in "owner" and all values of "ownedComment" are also contained in "ownedElement". The *union* construct specifies that the values of a property are contained in at least one of the properties that it is the union of. This construct is often used in combination with the *subsets* construct: one property contains the union of a number of other properties that in turn contain subsets of the first property. If the *subsets* or *union* construct does not specify what it is the subset or union of, then its complementing construct is required to specify this. The "ownedElement" property is an unspecified union and the complementing *subsets* construct at "ownedComment" (partly) specifies what it is the union of.

Another construct that is used a lot is that of *derived* properties, which are marked with a '/' (slash token). The properties "owner" and "ownedElement" are both marked derived. This means that their values are calculated from the values of other properties and no values are stored directly in a derived property itself. Derived properties are read-only.

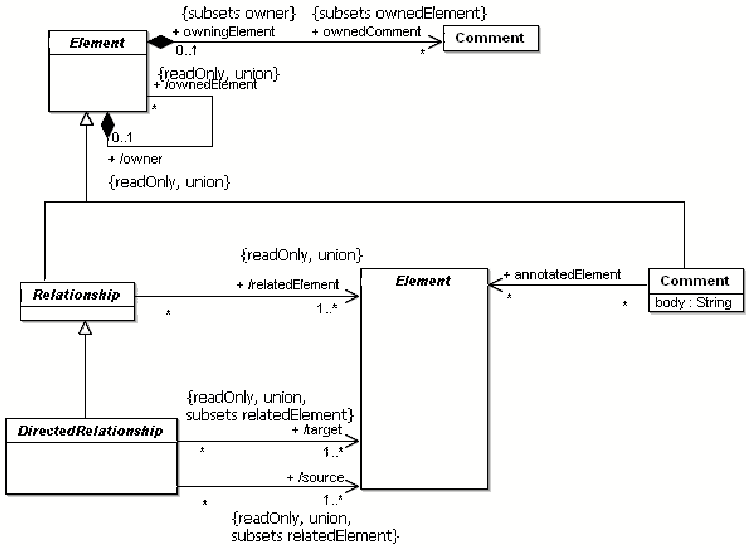

Figure 2.8: Root diagram of the UML Kernel package (source: [OMG05c]).

The MOF Model itself is expressed using MOF constructs: MOF Class and MOF Property are also instances of MOF Class. Hence, the MOF Model is its

own meta-model as well. The fact that the UML meta-model is expressed using constructs of the MOF Model, makes that the UML meta-model *conforms to* the MOF meta-model (see Fig. 2.6). Since the MOF meta-model is also expressed using constructs of the MOF Model, the MOF meta-model conforms to itself.

Note that while models expressed in MOF use associations to relate properties of classes to each other, there is no Association meta-class in the MOF Model. Associated properties are related toeach otherr using the "opposite" property. This means that only binary associations are allowed in MOF. Also note that MOF allows for operations to be defined on meta-classes. While MOF itself cannot express the behaviour of such operations, a framework that implements MOF can do this. MOF operations are mainly useful within model transformations, as will be made clear in the next section.

2.3.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [BSM+03] aims to comply with EMOF. Historically, EMF was intended as a proving ground for MOF 2.0. Hence, its meta-modelling language Ecore differs slightly from EMOF. Fig. 2.9 shows a simplified subset of the Ecore meta-model.

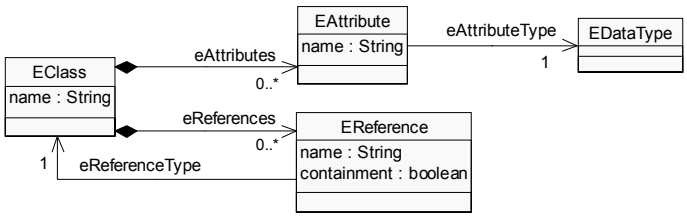

Figure 2.9: A simplified subset of the Ecore meta-model (source: [BSM+03]).

Note that Ecore prefixes an 'E' before all its meta-classes. This makes it easier to distinguish between Ecore classes and UML classes, for example. It also makes a distinction between EAttributes and EReferences, while EMOF always uses Property. The difference between EAttribute and EReference is that the type of an EAttribute is always a primitive type, such as string, boolean or integer, while the type of an EReference is always an EClass.

Ecore also supports "opposite" EReferences and EOperations, which are analogous to MOF "opposite" Properties and MOF Operations. They are not shown in Fig. 2.9, however. As an implementing framework, EMF allows the behaviour of EOperations to be specified in Java. EMF models are typically serialised in XMI [OMG05b].

In addition to the Ecore meta-modelling language, EMF provides a code generator framework that can generate Java code from Ecore models. This Java code represents EClasses, EAttributes, EOperations etc. through Java

classes, fields and methods. It is also through the generated Java code that the behaviour of EOperations can be defined: one simply fills in the body of the generated methods. The code generation of EMF goes further in that it can also generate a graphical editor for the generated model. A screenshot of the generated editor for the Ecore model itself (meta-level M3) is shown in Fig. 2.10.

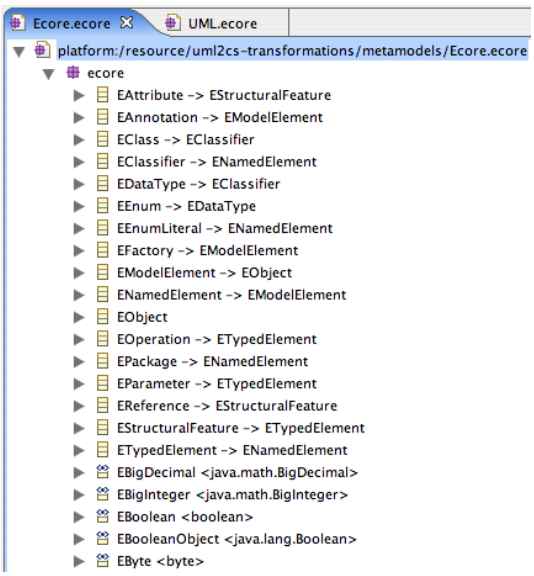

Figure 2.10: A screenshot of the Ecore editor.

2.3.3 The role of UML in the MDA

The Unified Modeling Language (UML) [OMG05c] is the standard modelling language for object-oriented design. It is called *unified* because it emerged from several other object-oriented modelling languages. It is not a *universal* modelling language in that it is suitable for modelling any (object-oriented) software system. UML is therefore presented as the standard modeling language for the MDA, while the MDA also allows the use of other modelling languages. The UML Profile mechanism in particular is used heavily in the MDA to introduce platform-specific annotations as well as to define platform-independent language constructs. We will therefore focus on the Profile mechanism.

The Profile mechanism provides a lightweight extension mechanism for UML that allows for *refining* the UML semantics. This means that each Profile has to respect the general UML semantics, but may introduce specific semantics that narrow down existing UML semantics. It uses Stereotypes to assign special meaning to designated model elements. Whenever a Stereotype is applied to a model element, this is shown by either an icon or the Stereotype

name between guillemets. Fig. 2.11 shows an example of a Stereotype application: the `<<Applet>>` Stereotype is applied to the `InstantMessagingClient` Class. The meaning of `<<Applet>>` is that `InstantMessagingClient` will be implemented as an applet that can be embedded in a web page or run on a mobile phone. Note that the general semantics for a UML class still apply; `InstantMessagingClient` is still a class, but with the additional, narrowed down semantics of an applet. The `<<Applet>>` Stereotype also provides the Tagged Value `appletInfo` of type `String`, which is set to "© 2007, Dennis Wagelaar" in this case.

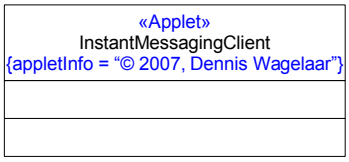

Figure 2.11: Applet Stereotype applied to a UML Class.

The structure of the Profile mechanism is shown in Fig. 2.12, which depicts the part of the UML *meta-model* that describes Profiles. It shows that Profile is a special kind of Package that can contain Stereotypes. A Stereotype is a special kind of Class that can function as an Extension to a meta-class, where an Extension is a special kind of Association.

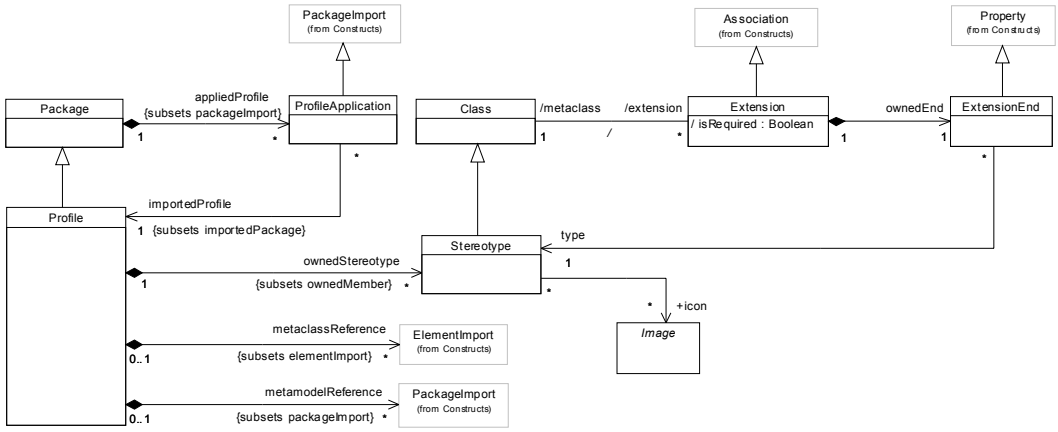

Figure 2.12: UML meta-model part for Profiles (source: [OMG05c]).

Fig. 2.13 shows the Applet Profile as used in the last example. The Profile itself is depicted as a tabbed box containing the other elements. The `<<Applet>>` Stereotype is depicted as a class with a `<<stereotype>>` tag, indicating that it is a special kind of class. `<<Applet>>` extends the UML meta-class `Class`. This Extension is depicted as an arrow. The meta-class `Class` is depicted as a class with a `<<metaclass>>` tag.

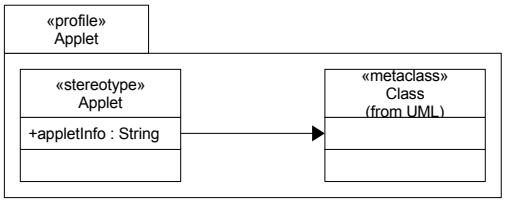

Figure 2.13: The Applet Profile.

Note that the part of the UML meta-model shown in Fig. 2.12 does not explicitly specify the structure of a Stereotype *application*. This is because Stereotype applications are represented as *instances* of the modelled Stereotype. Since instances of *model* elements (M1) only exist at the *instance* level (M0), the Profile must have a representation at the *meta-model* level (M2). That way, Stereotype instances can exist at the *model* level (M1).

Unfortunately, the semantics of Stereotypes and Stereotype applications are still under discussion in the research community [HSGP06] and cannot be considered well-defined. Stereotype applications as they are implemented in EMF *are* well-defined, however. That's why we choose to explain the EMF version of Stereotype applications later in this section.

2.3.4 Stereotype applications in EMF

There exists an EMF-based implementation of the UML meta-model, called Eclipse UML2. Historically, Eclipse UML2 was intended to test the UML 2.0 specification. The most notable difference between Eclipse UML2 and the UML specification is that Eclipse UML2 uses EMF's Ecore language to describe its meta-model. Eclipse UML2 and Ecore also do not share a common notion of meta-class in the way that the UML superstructure and MOF share the UML infrastructure notion of meta-class [OMG06d]. Eclipse UML2 and Ecore *do* share the meta-class EAnnotation. Fig. 2.14 shows how UML2's Element, which is the root meta-class of the UML2 meta-model, inherits from Ecore's EModelElement. Since each EModelElement can have EAnnotations, each Element can also have EAnnotations.

In order to achieve this sharing, the Eclipse UML2 meta-model *imports* the Ecore meta-model. This is possible because both meta-models exist at the M2 level (even though Ecore's meta-model *also* exists at the M3 level).

Now that we know how the Eclipse UML2 meta-model is built up, we can consider how Eclipse UML2 deals with profiles and stereotypes. The modelling of profiles and stereotypes follows the UML specification. Their solution for stereotype *applications* is different, however. After modelling an Eclipse UML2 profile, the Eclipse UML2 editor allows the developer to "define" the profile. This means that the editor creates an Ecore representation of the profile: the

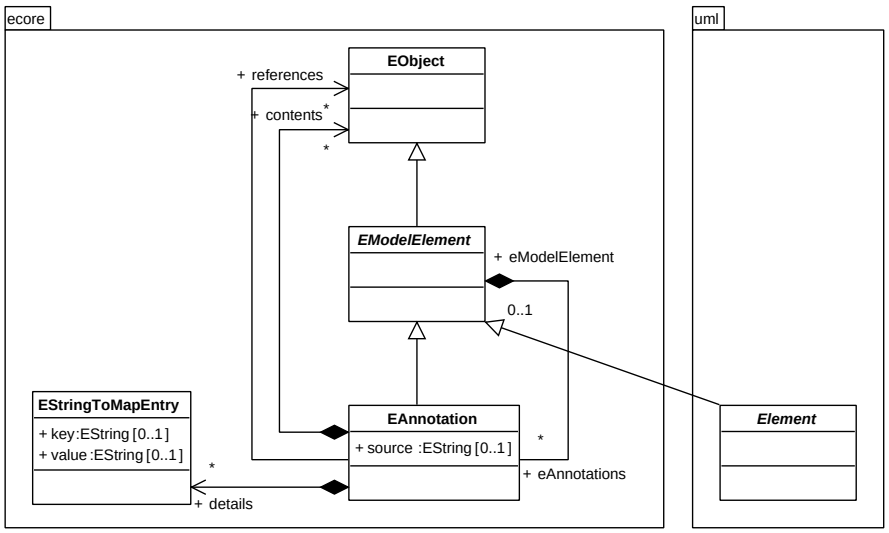

Figure 2.14: Eclipse UML2 and Ecore share the common notion of EAnnotations.

Eclipse UML2 representation of stereotypes at the M1 level is lifted up to an Ecore representation as EClasses at the M2 level. This approach is in line with the theory of *spanning objects* by Welty [WF94], where an element is modelled as both an instance and a class. Depending on the intended use, one refers to either the instance representation or the class representation of the element. In the case of Eclipse UML2, each Stereotype *instance* is mapped to an EClass *instance*. An EClass *instance* is a *class*, so we now have an instance representation and a class representation of the same element. Fig. 2.15 shows what the Applet profile looks like after the Eclipse UML2 editor has created an Ecore representation of the profile.

Note that Eclipse UML2 uses the common EAnnotation meta-class to embed an Ecore model inside a Eclipse UML2 model. The Applet profile contains an EAnnotation named "UML", indicated by a paperclip icon. EAnnotations can contain EObjects. Since EObject is the root EClass of the Ecore metamodel, EAnnotations can contain any Ecore element.

Since stereotypes are lifted from M1 to M2 level as EClasses, stereotype applications can now exist at the M1 level as instances of those EClasses. Any Eclipse UML2 model also exists at the M1 level, so it may include stereotype applications. Consider the UML Class diagram from Fig. 2.11. The Applet stereotype is applied to the InstantMessagingClient class. Fig. 2.16 shows an Object diagram of this situation. Black diamonds have been added to the links that represent containment: the "InstantMessengerModel" model contains the "im" package, which contains the "InstantMessagingClient" class. Note that

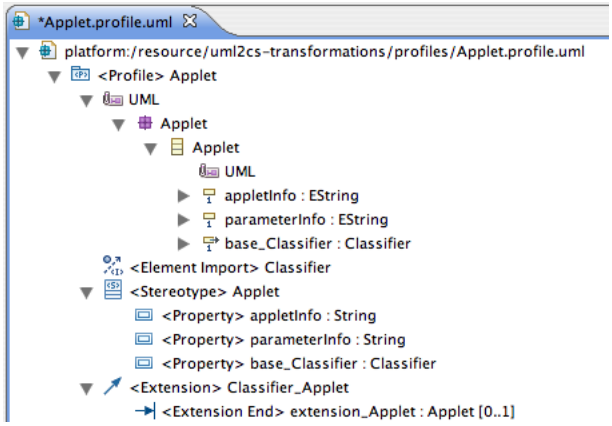

Figure 2.15: A screenshot of the Eclipse UML2 editor after the Applet profile was "defined".

the Applet stereotype application "app" is not contained anywhere, but is linked as a peer to the "InstantMessagingClient" class.

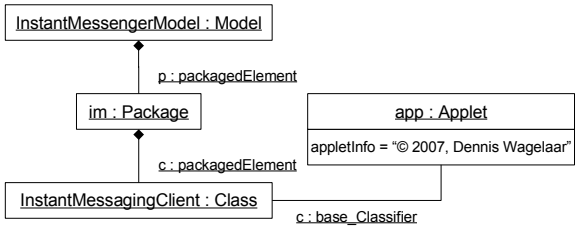

Figure 2.16: Object diagram of the Applet stereotype applied to the InstantMessagingClient class.

2.4 Model transformation

The key enabler of the MDA vision is model transformation. Model transformation itself depends heavily on meta-models. A meta-model defines the structure of a model and is used in model transformation to navigate models. Since the MDA focuses on MOF as the language of preference for defining meta-models, we will focus on model transformation languages that are built around MOF-compatible meta-modelling languages. We will discuss two MOF-based transformation languages:

- **MOF Query/View/Transformation (QVT)**
  The standard transformation language specification for MOF-based modelling languages.

- **ATLAS Transformation Language (ATL)**
  A QVT-like transformation language implemented in Java that can work with EMF models.

QVT exists as a specification without a reference implementation at the time of this writing[1]. ATL is one of the predecessors of QVT, has tool support and can work with EMF-based models. We have used ATL for the implementation of our case study. Both QVT and ATL use the Object Constraint Language (OCL) [Obj05] to express complex values.

The MDA advocates the use of multiple, successive refinement steps to transform a PIM into a PSM, which reduces the complexity of each transformation step (*divide-and-conquer*). We will demonstrate how successive PIM-to-PSM transformations can be expressed and employed through our instant messenger case study.

2.4.1 MOF Query/View/Transformation

Query/View/Transformation (QVT) [OMG05a] is the standard transformation language for MOF-based modelling languages, and is hence considered as a part of MOF. Fig. 2.17 shows an overview of the different parts of the QVT language and how they relate to each other.

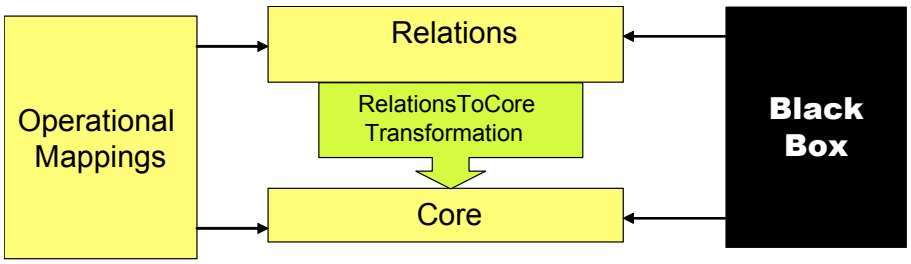

Figure 2.17: Relationships between QVT meta-models (source: [OMG05a]).

The Relations part is the declarative part of the QVT language. It declaratively expresses the relationship between the elements in its input models. The

[1] An EMF-based implementation of QVT is being developed within the Eclipse project.

Relations part maps to the Core part using the RelationsToCore transformation. The Core part is a small pattern-matching language and is mainly meant for use by transformation engines, while the Relations part is a human-usable language. Furthermore, QVT supports an imperative Operational Mappings part and a Black Box part for externally defined transformations. Operational Mappings can be used to implement one or more Relations from a Relations specification that are difficult to express in the Relations language. Black Box transformations serve as a "plug-in" implementation of a MOF operation. Black Box transformation code must be written in a programming language with MOF bindings, such that it is accessible from other QVT transformations. We will focus on the Relations part to explain the general philosophy behind QVT.

In the Relations language, a transformation between models is specified as a set of relations that must hold for the transformation to be successful. Each model in the transformation conforms to a model type, which is a specification of the kind of model elements that can occur in a conforming model. A model type is typically represented by a meta-model. The models in a transformation are named and are bound to a specific model type. An example Relations specification is:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) { ... }
```

In this declaration named "umlRdbms," there are two typed models: "uml" and "rdbms". The model named "uml" declares SimpleUML as its model type, and the "rdbms" model declares SimpleRDBMS as its model type.

A transformation can be invoked either to check two models for consistency or to modify one model to enforce consistency. A transformation invoked for enforcement is executed in a particular direction by selecting one of the models as the target. The target model may be empty, or may contain existing model elements to be related by the transformation. The execution of the transformation proceeds by first checking whether the relations hold. For relations for which the check fails, it attempts to make the relations hold by creating, deleting or modifying elements in the target model only, thus enforcing the relationship.

The relations in a transformation declare constraints that must be satisfied by the model elements. An example relation is:

```
relation PackageToSchema   /* map each package to a schema */
   {
      domain uml p:Package {name=pn}
      domain rdbms s:Schema {name=pn}
   }
```

In the example below, two domains are declared which will match elements in the "uml" and "rdbms" models. Domains represent the subject of the relation. Each domain specifies a simple pattern: a package with a name and a schema with a name. As a constraint, both "name" properties are bound to the same variable "pn", implying that they should have the same value.

The following example shows how **when** and **where** clauses can be used to refer to other relations:

```
relation ClassToTable   /* map each persistent class to a table */
   {
      domain uml c:Class {
            namespace = p:Package {},
            kind='Persistent',
            name=cn
      }
      domain rdbms t:Table {
             schema = s:Schema {},
              name=cn,
              column = cl:Column {
                    name=cn+'_tid',
                    type='NUMBER'},
              primaryKey = k:PrimaryKey {
                     name=cn+'_pk',
                     column=cl}
      }
      when {
         PackageToSchema(p, s);
      }
      where {
         AttributeToColumn(c, t);
      }
   }
```

The **when** clause specifies the conditions under which the relation needs to hold, so the relation "ClassToTable" needs to hold only when the "PackageToSchema" relation holds between the package containing the class and the schema containing the table. The **where** clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. Hence, whenever the "ClassToTable" relation holds, the relation "AttributeToColumn" must also hold.

A transformation contains *top-level* relations and *non-top-level* relations. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the **where** clause of another relation:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
   top relation PackageToSchema {...}
   top relation ClassToTable {...}
   relation AttributeToColumn {...}
}
```

"PackageToSchema" and "ClassToTable" are top level relations, whereas "AttributeToColumn" is a non-top-level relation.

2.4.2 ATLAS Transformation Language

The ATLAS Transformation Language (ATL) [JK06] historically served as a submission to the QVT Request For Proposals [OMG04b]. As a consequence, ATL is similar to QVT Relations, save some limitations: ATL transformations

are unidirectional. Output models are write-only and always start off as empty models. All navigation in ATL is done on read-only input models. QVT checking transformations are typically implemented as ATL *queries*, while enforcing transformations are represented in ATL as *modules*.

ATL modules

An ATL transformation module has a number of input models and typically one output model. It contains a number of *rules* that define the mapping from source elements to target elements. ATL has two kinds of rules: *matched* rules and *called* rules. These compare to QVT *top-level* relations and *non-top-level* relations in that matched rules are automatically triggered, while called rules must be invoked from a matched rule. The following example ATL module copies a UML Model element to another UML Model element:

```
module UML2Copy;

create OUT : UML2 from IN : UML2;

rule Model {
    from s : UML2!uml::Model
    to t : UML2!uml::Model (
        name <- s.name,
        visibility <- s.visibility,
        viewpoint <- s.viewpoint)
}
```

The UML2Copy module has one output model named "OUT" of model type "UML2" and one input model "IN", which is also of model type "UML2". The model type "UML2" corresponds to the Eclipse UML2 meta-model in this case, which is normally specified in the Eclipse "Run..." dialogue window. The transformation module has one *matched* rule named "Model". Since ATL transformations are unidirectional, ATL rules don't have a **domain** construct like QVT relations. Instead, ATL rules have a **from** part and a **to** part. The **from** part specifies which model elements from the input model(s) trigger the matched rule. The **to** part creates one or more model elements in the output model. In the example, any instance of the meta-class "uml::Model" from the "UML2" meta-model triggers the rule, where the "uml::" prefix specifies that the "Model" meta-class is inside the "uml" package. ATL uses '`<-`' to specify assignment: the Model copy has the same name, visibility and viewpoint values as the original Model instance. The following example shows how multiple matched rules interact and how called rules work:

```
module UML2ExtendedCopy;

create OUT : UML2 from IN : UML2;

rule Model {
    from s : UML2!uml::Model
    to t : UML2!uml::Model mapsTo s (
        name <- s.name,
        visibility <- s.visibility,
```

```
        viewpoint <- s.viewpoint,
        packagedElement <- s.packagedElement)
}

rule Package {
    from s : UML2!uml::Package (
        s.oclIsTypeOf(UML2!uml::Package))
    to t : UML2!uml::Package mapsTo s (
        name <- s.name,
        visibility <- s.visibility,
        packagedElement <- s.packagedElement
            ->including(thisModule.AddClass(s))),
}

rule AddClass(p : UML2!uml::Package) {
    to t : UML2!uml::Class (
        name <- p.name + 'Info')
    do {
        t;
    }
}
```

The "Model" rule now includes an assignment of the "packagedElement" property. The "packagedElement" property refers to a collection of packaged model elements in the source model. Each of those model elements may separately match against a rule in the transformation module. Normally, the *target* element of the "Model" rule is supposed to contain the *target* "packagedElement" elements, just like the source "Model" element contains the source "packagedElement" elements. ATL automatically translates assignments of source elements to their target element counterparts whenever those source elements trigger a matched rule in the transformation module.

This kind of source-element-to-target-element resolution is made explicit by the **mapsTo** statement. The **mapsTo** statement specifies which target element maps back to which source element and is used to translate an assignment of source values to an assignment of target values: the target "packagedElement" collection in the "Model" rule will not contain the values of "s.packagedElement", but rather the collection of target elements that map to the values of "s.packagedElement".

The "Package" rule copies all instances of "uml::Package" that satisfy the additional condition "`s.oclIsTypeOf(UML2!uml::Package)`". This additional condition is necessary to prevent the rule from triggering against subclasses of "uml::Package", such as "uml::Model".

For each "uml::Package", a "uml::Class" is added by the "AddClass" *called* rule. This rule is triggered by the assignment of "packagedElement" in the "Package" rule: each "packagedElement" is a copy of the source "packagedElement" augmented with the additional class that is returned by the "AddClass" rule.

Since called rules are always explicitly invoked, they have no **from** part. All input elements are passed as operation-style parameters instead. The "AddClass" rule takes a "uml::Package" as a parameter. It creates a new

"uml::Class" with its name set to the name of the input "uml::Package" augmented with the string '`Info`'.

In ATL, each rule can also have an imperative **do** part that consists of a number of assignment or OCL statements. The **do** part is also used to specify a return value for a called rule. "AddClass" returns the value of the "uml::Class" it generates. This return value is used in the "Package" rule to add the generated "uml::Class" to the "packagedElements".

Helpers

ATL transformations can use *helpers* to encapsulate complex OCL expressions. Helper *methods* are used to encapsulate an OCL expression, possibly applied to a number of parameters. Helper *attributes* are used to store the value of an OCL expression and cannot take parameters. Below is an example of both a helper attribute and a helper method:

```
helper def : inElements : Set(UML2!ecore::EObject) =
    UML2!ecore::EObject.allInstancesFrom('IN');

helper context UML2!ecore::EObject def : isPackage() : Boolean =
    self.oclIsTypeOf(UML2!uml::Package);
```

The "inElements" helper attribute holds a set of "ecore::EObject" instances. As was already stated in the previous subsection, the UML2 meta-model import the Ecore meta-model. As a result, the UML2 meta-model consists of the "ecore" package and the "uml" package. The "inElements" helper attribute uses the "allInstancesFrom" method to retrieve all instances of the "ecore::EObject" meta-class from the "IN" model.

The "isPackage" helper method can be invoked on all "ecore::EObject" instances (its **context**), takes no additional parameters and returns a boolean value. Instead of writing "`p.oclIsTypeOf(UML2!uml::Package)`" for a given ecore::EObject p, one can now write "`p.isPackage()`".

ATL queries

The main difference between ATL queries and modules is that queries don't have an output model. They also don't have transformation rules, as is shown in the following example query:

```
query UML2toJava = UML2!uml::Classifier.allInstances()->collect(e |
    if e.ignore() then true
    else e.toFileString().writeTo(e.pathName())
    endif);
```

Queries can take a number of specified input meta-models and a number of implicit input models. The "UML2toJava" query has "UML2" as its specified input meta-model and can take one or more input models that conform to "UML2". There is no explicit reference to the input model(s): instead, model

elements are found via their meta-class, which is "uml::Classifier" in the example. The query collects specific values for all classifier instances. It uses a number of helper methods, which are not specified here, to calculate these values. Since queries are not typed, any value is allowed. In fact, the end result value of a query is never used: ATL includes a number of special helpers that perform actions as printing text or writing to file. The example query is meant to generate code for each classifier, which is then written to the correct path.

ATL libraries

Both ATL modules and queries can make use of ATL libraries. A library can contain helper methods that are reusable over multiple modules and/or queries. A module can import a library with the **uses** statement, which comes right after the **create** clause:

```
module UML2LibraryTransformation;

create OUT : UML2 from IN : UML2;

uses UML2;
```

For queries, the **uses** statement comes right after the **query** statement:

```
query UML2toJava = UML2!uml::Classifier.allInstances()->collect(e |
    if e.ignore() then true
    else e.toFileString().writeTo(e.pathName())
    endif);

uses UML2;
```

The "UML2" library itself looks like this:

```
library UML2;

uses Strings;

helper context String def : type() : UML2!uml::Type =
  UML2!uml::Type.allInstances()->select(c|c.umlQualifiedName()=self)->first();

helper context UML2!uml::NamedElement def : umlQualifiedName() : String =
  self.qualifiedName('::');

helper context UML2!uml::NamedElement
def : qualifiedName(separator : String) : String =
  if self.owner.oclIsTypeOf(UML2!uml::Package) then
    self.owner.qualifiedName(separator) + separator + self.name
  else
    self.name
  endif;
```

Note that libraries themselves can also import other libraries, such as "Strings" in this case. The can also refer to specific meta-models, such as "UML2" in this case.

ATL superimposition

While ATL transformation modules and queries are normally run one at a time, it is also possible to *superimpose* several transformation modules on top of each other. The end result is a transformation module that contains the union of all transformation rules and all helpers, where it is possible for a transformation module to *override* rules and helpers from other transformation modules. Fig. 2.18 shows an example of a typical use case for superimposition: the transformation rules of the UML2Copy module are reused and overridden where necessary by the UML2Profiles module.

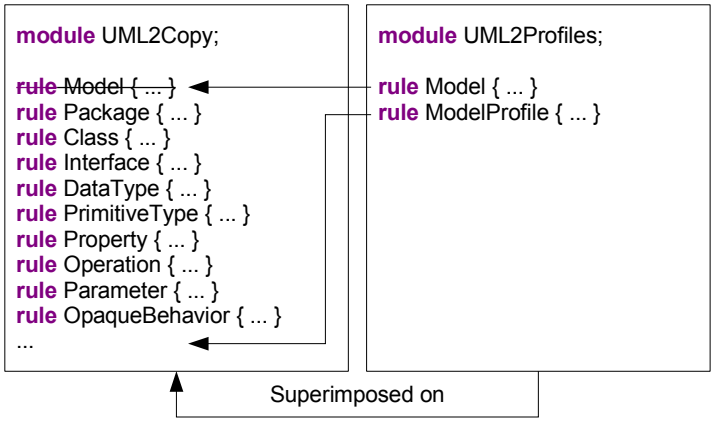

Figure 2.18: ATL superimposition example.

The UML2Copy transformation module includes a transformation rule for every meta-class instance it must copy. This amounts to approximately 200 rules for the entire UML2 meta-model[2]. The UML2Profiles transformation module applies a profile to the "uml::Model" instance, provided it was not yet applied. All other elements should just be copied. Instead of copying-and-pasting the 200 rules from UML2Copy, the UML2Profiles module is superimposed on the UML2Copy module. It overrides the "Model" rule, which copies each "uml::Model" instance, by a version that checks that the profile we want to apply has already been applied. It also introduces a new rule "ModelProfile", which checks that the profile we want to apply has **not** been applied and then applies the profile. The resulting transformation is shown in Fig. 2.18.

Note that superimposition is a load-time construct: there is no real transformation module that represents the result of superimposing several modules on top of each other. Instead, several modules are simply *loaded* on top ofeach otherr, overriding existing rules and adding new rules. Since superimposition

[2]The UML2Copy transformation is generated from the UML2 meta-model by a *higher-order* transformation: a transformation that generates another transformation.

does not require re-compilation of the involved modules, it can be used to scale up the ATL compiler to much bigger transformation modules. Superimposition allows developers to partially compile their transformations by splitting them up in multiple modules. For the developer, this partial compilation speeds up the development process as it requires less resources.

Superimposition is a relatively new feature that we've contributed to ATL. It is described in more detail in [Wag08].

2.4.3 PIM-to-PSM refinements

The MDA uses model transformation to transform PIMs into PSMs. This can be done in one transformation, but often you will want to tackle one transformation problem at a time, such as generating accessor methods or implementing an observer pattern. That's why MDA advocates the use of successive refinement steps instead of one single transformation. We will demonstrate how successive PIM-to-PSM refinement transformations are implemented in our instant messenger case study.

Our case study employs a number of successive refinement transformations to achieve a PSM like the one shown in Fig. 2.5. Each refinement transformation adds some elements to the model for each model element that it triggers on. The model transformation that introduces accessor methods (getters and setters) triggers on all public properties, for example. We also use a number of alternative refinement transformations, such that we can target different platforms. Table 2.1 shows the available refinement transformations, where each row may contain several alternative transformations, separated by a '|'. The transformations are presented in order of execution, in that one transformation from each row must be executed after a transformation from the previous row. The transformations are written in ATL and their complete source code can be found at `http://ssel.vub.ac.be/viewvc/UML2CaseStudies/uml2cs-transformations/`.

Below is a description of each model transformation:

UML2Profiles: The UML2Profiles transformation applies all UML profiles that are necessary to express our PSM. This currently comprises only the "Accessors" profile, which allows to specify "accessor" relationships between properties and their accessor methods. The "accessor" relationship is modelled as a UML Dependency relationship that is stereotyped as "accessor".

UML2Accessors: The UML2Accessors transformation introduces accessor methods for each public, navigable property. It uses the "Accessors" profile to track which accessors have been introduced for which property and it also takes into account property multiplicity, uniqueness and ordering. The UML2Accessors transformation uses an ATL library for the

UML2Profiles
UML2Accessors + Java1 | UML2Accessors + Java2
UML2Observer + Java1 | UML2Observer + Java2 | UML2JavaObserver + Java1 | UML2JavaObserver + Java2
UML2AbstractFactory
UML2Singleton
UML2Applet | UML2MIDlet
UML2AsyncMethods
UML2DataTypes + Java1 | UML2DataTypes + Java2

Table 2.1: Available model transformations

Java language mappings; either the combination of the "JavaMappings" and "Java1" library are used, or the combination of "JavaMappings" and "Java2". "Java1" is responsible for mapping collection data types to `java.util.Vector`, which is available in all Java platforms. "Java2" maps collections to `Java.util.List` and `java.util.Set`, which are only available in the Java platforms that support the Java 2 Collections framework.

UML2Observer: The UML2Observer transformation introduces an implementation of the observer design pattern. It uses the "Observer" profile to find out which elements play which role in the observer pattern. The "Observer" profile must be applied by the software engineer and offers the the stereotypes "Observer", "Observable" and "subscribes". The "Observer" stereotype applies to classes whose instances can observe other objects. The "Observable" stereotype applies to classes whose instances can be observed by other objects. The "subscribes" stereotype applies to associations via which "Observer" objects can subscribe to "Observable" objects. The UML2Observer transformation introduces all the event triggering infrastructure and adapts the setter methods. Event handling operations must be introduced by the software engineer and are named according to the property one wants to observe: if the "name" property of an observable class must be observed, an "onNameChange" operation must be defined in the observer class. The model transformation makes sure that this handler operation is triggered. UML2Observer needs to work with the accessor operations created by and the data types used by the UML2Accessors transformation, which means it also requires either the "JavaMappings" + "Java1" or the "JavaMappings" + "Java2" library combination.

UML2JavaObserver: The UML2JavaObserver transformation introduces similar functionality as the UML2Observer transformation, but uses the existing `java.util.Observer` API to implement it. It also uses Java reflection to find observer handler methods, which allows the software engineer to add new handlers after the transformation has been executed. Neither `java.util.Observer` nor Java reflection are available in J2ME MIDP, for example, so the result of this transformation will not work on all Java platforms.

UML2AbstractFactory: The UML2AbstractFactory transformation introduces an implementation of the abstract factory design pattern. It uses the "AbstractFactory" profile to find out which elements play which role in the abstract factory pattern. Same as the "Observer" profile", the "AbstractFactory" profile must be applied by the software engineer. Available stereotypes are "AbstractFactory", "ConcreteFactory" and "product". The "AbstractFactory" stereotype applies to classifiers that represent that abstract factory interface for the design pattern. The "product" stereotype is applied to dependency relationships that point to the abstract product classes or interfaces. The "ConcreteFactory" stereotype applies to classes that represent a factory implementation. The "product" dependency relationships are also used here, but this time to mark the concrete products that are created by the factory implementation. The model transformation introduces the "create" operations, based on the abstract product names (e.g. "createContactListView" for an abstract product interface named "ContactListView"). It also generates the implementation for these operations for each concrete factory.

UML2Singleton: The UML2Singleton transformation introduces an implementation of the singleton design pattern. It uses the "Singleton" profile to do this, which simply offers the "Singleton" stereotype to mark singleton classes. The model transformation introduces a "getInstance" operation for each singleton class. It does not hide the constructor operation, since this creates problem in particular cases, such as live GUI editors and Java applets.

UML2Applet: The UML2Applet transformation introduces an implementation for classes that are stereotyped as "Applet". The implementation is based on the `java.applet.Applet` class. It uses the "Applet" profile, which must be applied by the software engineer.

UML2MIDlet: The UML2MIDlet transformation also introduces similar functionality as the UML2Applet transformation, except that its implementation is based on the `javax.microedition.midlet.MIDlet` class. In addition, a special adapter class is introduced, such that "Applet"

classes can use the special operations for standard Java applets ("start", "stop", etc.) instead of the special MIDlet operations.

UML2AsyncMethods: The UML2AsyncMethods transformation wraps the implementation of all operations stereotyped as "asynchronous" inside a `java.util.Thread`, such that each invocation of that operation becomes asynchronous. It uses the "Async" profile to to do this, which must be applied by the software engineer.

UML2DataTypes: The UML2DataTypes transformation replaces all OCL data types into Java classes, interfaces and primitive types. It also deals with multiplicity, uniqueness and ordering constraints by translating to the correct Java collection type. That's why this transformation also requires either the "JavaMappings" + "Java1" or the "JavaMappings" + "Java2" library combination. The OCL data types are provided in a separate UML model, that can be imported by the software engineer.

Most of these refinement transformations add relatively small platform dependencies to the model, generally in the form of required Java API (e.g. `java.applet.Applet` or `java.util.List`). Considered alone, these separate platform dependencies allow for more possible target platforms than when they are considered together. For example, `java.applet.Applet` is available on all Java platforms that support the AWT GUI toolkit, which includes everything from Personal Java to Java 6 SE. Likewise, `java.util.List` is available on all Java platforms that support the Java 2 Collections framework. When combined, these platform dependencies narrow down the target platform to all Java platform that support both AWT **and** the Java 2 Collections framework.

This example may seem a bit contrived, as all platforms that support the Java 2 Collections framework also support AWT, but it illustrates the point that a single refinement transformation introduces only small, local platform dependencies for each element it transforms. Also note that in reality, there are different versions of AWT and the Java 2 Collections framework, such that satisfying even these two platform dependencies is not trivial.

Even though Table 2.1 shows the possibility for many combinations, care must be taken that a meaningful combination is chosen. There are combinations of refinement transformations with platform dependencies that no currently existing platform can satisfy. For example, none of the currently existing Java platforms supports both `javax.microedition.midlet.MIDlet` **and** `java.util.Observer`. Also the choice for the "Java1" or "Java2" mapping library must be consistent: the same choice must be made for all relevant transformations, such that these transformations assume the same collection types.

2.5 Summary

In this chapter we have explained the Model Driven Architecture (MDA). We have positioned the MDA as an approach to software development that aims to lift the feedback in the development lifecycle up from code level to analysis and design level. It does this by means of automated transformations from abstract, platform-independent software descriptions to concrete, platform-specific software descriptions. In the MDA, software descriptions take the form of models. We have explained the roles of the various kinds of models used by the MDA. We have also explained how the MDA uses meta-modelling and model transformation technologies to achieve the transitions between levels of abstraction. We have explained the Eclipse Modeling Framework (EMF) and the ATLAS Transformation Language (ATL) in particular, since our case study builds on these technologies. Finally, we have demonstrated how PIM-to-PSM refinement transformations can be implemented and how platform dependencies are introduced.

The next chapter discusses ontologies and the OWL language [SWM04] in particular. The OWL ontology language forms the basis for our platform models.

Chapter 3

Ontologies

3.1 Introduction

The term "ontology" has many interpretations. We base ourselves on Gruber's interpretation, which is the one that is generally used for ontologies in the context of software systems:

> "An *ontology* is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an ontology is a systematic account of Existence. For knowledge-based systems, what "exists" is exactly that which can be represented." [Gru93]

Ontologies are typically used to share knowledge among software systems and serve as a common, controlled vocabulary in which shared knowledge is – often formally – represented. Hence, ontologies are very useful for *domain modelling*, in which a conceptualisation of a particular domain is given.

In the past, many formalisms were used to represent an ontology. Currently, a standard language for expressing ontologies has emerged and current research on ontologies is carried out in the scope of this language: the Web Ontology Language (OWL) [SWM04]. The remainder of this chapter explains how OWL is used to express ontologies.

OWL has been developed mainly to support the vision of the Semantic Web[1], in which the information that is accessible via the Web is related to each other via ontologies. OWL has been designed with automatic reasoning in mind. Automatic reasoning enables separate parts of the Semantic Web to be automatically related toeach otherr.

[1] http://www.w3.org/2001/sw/

OWL serves as a general ontology language as well, since it supports all the necessary concepts, such as *classes*, *properties*, *individuals* and *relationships* between these individuals. OWL is built on top of the Resource Description Framework (RDF) language [BG04] and provides three increasingly expressive sublanguages (from [SWM04]):

- *OWL Lite* supports those users primarily needing a classification hierarchy and simple constraint features. For example, while OWL Lite supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives. OWL Lite should also provide a quick migration path for thesauri and other taxonomies.

- *OWL DL* supports those users who want the maximum expressiveness without losing computational completeness (all entailments are guaranteed to be computed) and decidability (all computations will finish in finite time) of reasoning systems. OWL DL includes all OWL language constructs with restrictions such as type separation (a class can not also be an individual or property, a property can not also be an individual or class). OWL DL is so named due to its correspondence with *description logics* [BCM+03]. OWL DL was designed to support the existing Description Logic business segment and has desirable computational properties for reasoning systems.

- *OWL Full* is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support every feature of OWL Full.

We will focus on OWL DL where the different OWL versions deviate from each other, since it is the most expressive version of OWL that still allows for automatic reasoning. The remainder of this section discusses the OWL DL language constructs.

3.2 Simple named classes

Domain concepts are generally represented as simple named *classes*, which can have subclasses like in object-oriented programming. The subclass relationship is also called *subsumption* in OWL. Every *individual* – or class instance – in

OWL is an instance of the predefined class "owl:Thing"[2]. Each user-defined class is implicitly a subclass of "owl:Thing": all instances of each user-defined class are also instances of "owl::Thing". Domain-specific root classes are defined by simply declaring a named class. OWL also predefines the empty class, "owl:Nothing".

Let's consider the example of the domain of wines from [SWM04]. Fig. 3.1 shows a basic classification for this domain.

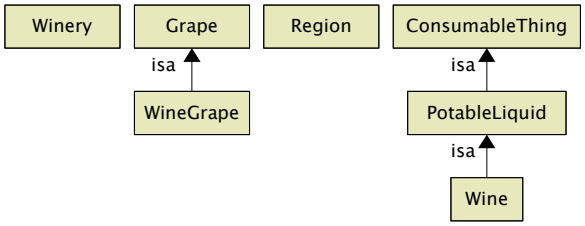

Figure 3.1: An example classification of wine-related concepts in OWL.

This simple wines ontology introduces three root classes, "Winery", "Grape", "Region" and "ConsumableThing", all of which are implicitly a subclass of "owl:Thing". In addition, "Grape" has a subclass "WineGrape" and "ConsumableThing" has a subclass "PotableLiquid", which in turn has a subclass "Wine". Subclassing is denoted via a directed arrow with the label "isa".

3.3 Individuals

Now that we can describe classes, we can also describe their members. These class members – or instances – are called *individuals* in OWL. Individuals are generally declared as being a member of a specific class, as shown in Fig. 3.2.

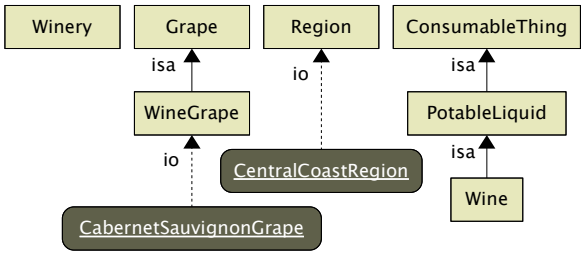

Figure 3.2: Some example OWL individuals in the wine domain.

[2] "owl:" refers to the XML namespace of the OWL language, which is in the form of an XML schema

The extended wine ontology adds two individuals, "CentralCoastRegion" and "CabernetSauvignonGrape", which are instances of "Region" and "WineGrape", respectively. The *instance of* relationship is denoted by a directed, dashed arrow with the label "io". We have decided that "CentralCoastRegion" (a specific area) is member of "Region", the class containing all geographical regions. We have also decided that "CabernetSauvignonGrape" is an individual because it denotes a single grape varietal.

The choice of representing something as a class or an instance is an important one and is often difficult to make. The OWL guide has the following to say about this:

> "There are important issues regarding the distinction between a class and an individual in OWL. A class is simply a name and collection of properties that describe a set of individuals. Individuals are the members of those sets. Thus classes should correspond to naturally occurring sets of things in a domain of discourse, and individuals should correspond to actual entities that can be grouped into these classes.
> In building ontologies, this distinction is frequently blurred in two ways:
>
> - *Levels of representation:* In certain contexts something that is obviously a class can itself be considered an instance of something else. For example, in the wine ontology we have the notion of a "Grape", which is intended to denote the set of all *grape varietals*. "CabernetSauvingonGrape" is an example instance of this class, as it denotes the actual grape varietal called Cabernet Sauvignon. However, "CabernetSauvignonGrape" could itself be considered a class, the set of all actual Cabernet Sauvignon grapes.
>
> - *Subclass vs. instance:* It is very easy to confuse the instance-of relationship with the subclass relationship. For example, it may seem arbitrary to choose to make "CabernetSauvignonGrape" an individual that is an instance of "Grape", as opposed to a subclass of "Grape". This is not an arbitrary decision. The "Grape" class denotes the set of all *grape varietals*, and therefore any subclass of "Grape" should denote a subset of these varietals. Thus, "CabernetSauvignonGrape" should be considered an *instance of* "Grape", and not a subclass. It does not describe a subset of grape varietals, it *is* a grape varietal." [SWM04]

The choice between classes and individuals is also discussed in detail by Welty and Ferrucci in [WF99]. How an ontology should be built with regard to classes and individuals depends on the intended usage. The issues regarding

the choice for classes or individuals also bring up one major difference between OWL Full and OWL DL. OWL Full allows the use of classes as instances and OWL DL does not.

3.4 Simple properties

Properties allow us to assert general facts about the members of classes and specific facts about individuals. A *property* is a binary relation. Two types of properties are distinguished:

- *Datatype properties:* relations between instances of classes and primitive data types (e.g. integer or string).
- *Object properties:* relations between instances of two classes.

Each property has a domain and a range. Subproperties can be defined for a property, which restrict the domain and/or range. Fig. 3.3 shows an example object property and how it applies to individuals.

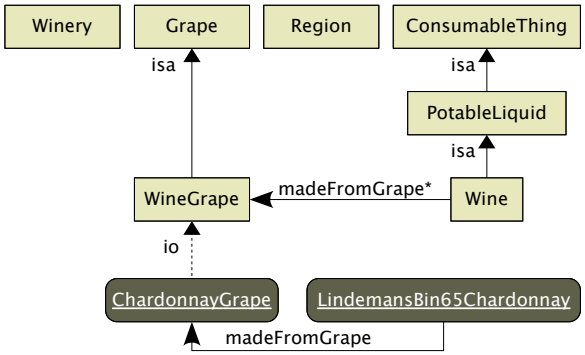

Figure 3.3: An example OWL property in the wine domain.

The property "madeFromGrape" has a domain of "Wine" and a range of "WineGrape", which means it relates instances of the class "Wine" to instances of the class "WineGrape". We've used a '*' in Fig. 3.3 to indicate that "madeFromGrape" can relate "Wine" instances to multiple "WineGrape" instances. This is the default behaviour in OWL. It is possible to restrict a property to relate an individual to only one other individual through a *functional* property, as is discussed later.

The use of range and domain information in OWL is different from type information in a programming language. Types are used to check consistency in a programming language, whereas in OWL, a range may *also* be used to infer a type. Consider the "LindemansBin65Chardonnay" from Fig. 3.3. We

can infer that "LindemansBin65Chardonnay" is an instance of "Wine" because the domain of "madeFromGrape" is "Wine".

Properties can be arranged in a hierarchy, just like classes. Fig. 3.4 shows an example of a subproperty as well as a *transitive* property.

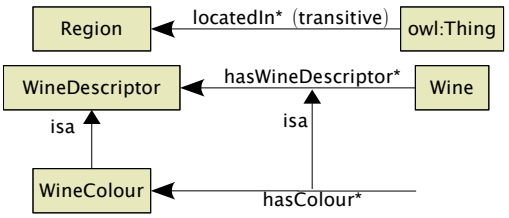

Figure 3.4: An example of an OWL subproperty and a transitive property.

The "WineDescriptor" class refers to a wine's colour and components of taste, including sweetness, body, and flavour. A "Wine" is related to "WineDescriptor" via the "hasWineDescriptor" property. "hasColour" is a subproperty of the "hasWineDescriptor" property, as shown by the "isa" relationship. While its range is further restricted to "WineColour", no domain has been specified for "hasColour": the domain is inherited from "hasWineDescriptor". The subproperty relationship means that every "hasColour" property value of a "Wine" individual also counts as a "hasWineDescriptor" property value.

The "locatedIn" property from Fig. 3.4 describes in which "Region" something is located. It's domain is "owl:Thing", which means that any individual can have a "locatedIn" property value, including instances of "Region". The "locatedIn" property is *transitive*: if an individual "Brussels" is located in "Belgium" and "Belgium" is located in "Europe", then "Brussels" is also located in "Europe".

Properties can have several other characteristics, such as being *symmetric*, *functional* or an *inverse of* another property. Fig. 3.5 shows several examples.

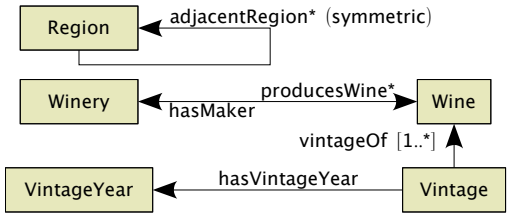

Figure 3.5: Example of symmetric, functional and inverse properties.

The "adjacentRegion" property is *symmetric*, which means that if a region "Flanders" is adjacent to a region "Wallonia", then "Wallonia" is adjacent to "Flanders" as well. The "hasVintageYear" property is a *functional*

property, which means that any domain value maps to a single range value: each "Vintage" can only have one "VintageYear". The "producesWine" and "hasMaker" properties are inverse properties, which means that if and only if a winery "ChateauMigraine" produces the "GrandCervigenic" wine, then "GrandCervigenic" has "ChateauMigraine" as its maker. The "Vintage" and "Wine" classes are related to each other through the "vintageOf" property. "vintageOf" has a *cardinality constraint* applied to it: a "Vintage" instance is a "vintageOf" at least one "Wine". This is depicted by the "[1..*]" label.

> Relating "Vintage" to "Wine" through a property is not an obvious choice: we could have modelled "Vintage" as a subclass of "Wine" with the thought that only some wines classify as a vintage. This is wrong, however, since the "Wine" class denotes the set of all *varieties* of wine, as has been discussed before. A "Vintage" is not a *variety*, but refers to a particular *year* of that variety. This issue of *subclass vs. property* is different from the *subclass vs. instance* issue, since here the distinction is not *subset* versus *identity*, but whether the classes involved are *disjoint*. OWL allows for explicit specification of disjoint classes to help detect this issue. This issue and other issues are discussed in detail by Rector et al. in [RDH+04].

3.5 Property restrictions

It is possible to further constrain the range of a property with *property restrictions*. Property restrictions always apply to a specific property and they come in several types: "allValuesFrom", "someValuesFrom", "cardinality" and "hasValue". A property restriction can be treated as an anonymous OWL class, which means that it is possible to define another OWL class as a subclass of a property restriction. Property restrictions must only hold in the context of their subclasses, which may only be a small part of the entire property domain. Fig. 3.6 shows an example of an "allValuesFrom" and a "someValuesFrom" property restriction, which are depicted using the '$\forall$' and the '$\exists$' symbols. We will follow the notation used in the Protégé ontology editor[3] where possible, which is one of the most popular ontology editors.

The "Winery" class is defined as a subclass of the "$\forall$ locatedIn WineRegion" property restriction, which is indicated by the '$\sqsubseteq$' symbol in front of the restriction. This means that the "locatedIn" property has been restricted for the "Winery" domain: all wineries must be located in wine regions. Note that

[3] http://protege.stanford.edu/

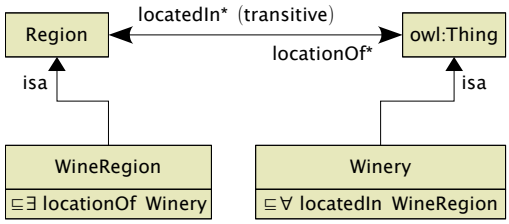

Figure 3.6: Example of "allValuesFrom" and "someValuesFrom" property restrictions.

this restriction does not *require* a winery to be located in a wine region: only *if* a winery has locations, those locations are wine regions.

The "WineRegion" class restricts the "locationOf" property as "∃ locationOf Winery": a wine region is the location of at least one winery. This restriction does not require all things located in a wine region to be wineries, but it *does* require at least one thing to be a winery.

> This example exhibits a peculiar side-effect, since "locatedIn" and "locationOf" are inverse properties. If we enforce the "∃ locationOf Winery" restriction for the "WineRegion" class, then this has consequences for the "locatedIn" property as well. If we require all wine regions to be the location of at least one winery, then the inverse must hold as well: for all regions that are the location of a thing, that thing is located in that region. So, for all wine regions that are the location of a thing, that thing is located in that wine region **and** at least one of those things is a winery!
> This is *consistent* with the "∀ locatedIn WineRegion" restriction for the "Winery" class, since it allows a winery to be located in a wine region (in fact, wineries must always be located in a wine region). If we change the "Winery" restriction to "∀ locatedIn ¬WineRegion", the ontology would be *inconsistent*: wine region *requires* at least one winery to be located in it, while no wineries may be located in wine regions! This demonstrates that the definition of property restrictions requires extra care in the presence of inverse properties.

We have already seen cardinality constraints in Fig. 3.5: "vintageOf" had a minimum cardinality of '1'. It is also possible to specify a maximum cardinality, as well as an *exact* cardinality. Cardinality expressions with a value of 0 or 1 are allowed in OWL Lite, whereas OWL DL supports any positive integer value.

The "hasValue" property restriction allows us to specify classes based on the existence of particular property values. An individual will be a member

of such a class if at least one of its property values is equal to the specified property value. Fig. 3.7 shows an example of a "hasValue" property restriction, which is depicted using the '∋' symbol.

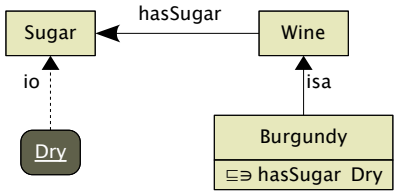

Figure 3.7: Example of a "hasValue" property restriction.

The property restriction "∋ hasSugar Dry" declares that all "Burgundy" wines are dry. That is, their "hasSugar" property must have at least one value that is equal to "Dry".

3.6 Ontology mapping

Ontologies are expected to be defined by many different people and/or organisations. Hence, these different ontologies may present different views of the same domain, which can result in some overlap. OWL provides two constructs to declare such overlap: *equivalence* for classes and properties and *identity* for individuals.

Consider the two equivalence examples in Fig. 3.8, which are depicted using the '≡' symbol and a different colour for the affected class.

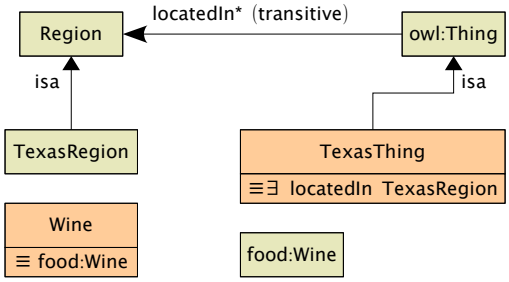

Figure 3.8: Example of equivalence.

The "Wine" class has been declared equivalent to the "food:Wine" class from the food ontology, which means that all instances of "Wine" are to be considered instances of "food:wine" as well and vice versa. We use a different colour for a class that has an equivalence declaration to indicate that the equivalence relationship often has significant consequences for the classification of other classes and individuals. The class hierarchy changes in that all sub-

and superclasses of "food:Wine" become sub- and superclasses of "Wine" as well and vice versa. The changes in class hierarchy then propagate to the individuals.

Equivalence declarations can also be used on property restrictions (since they are in effect anonymous classes). "TexasThing" has been declared equivalent to the property restriction "∃ locatedIn TexasRegion". This means that each individual that has a "TexasRegion" value in its "locatedIn" property (i.e. each individual that is located in Texas) is equivalent with "TexasThing". This may again have consequences for classification: all individuals that are declared instances of "owl:Thing" **and** are located in "TexasRegion" are now considered to be instances of "TexasThing" as well.

> The equivalence relationship should be used with care, as its use may easily result in inconsistent ontologies or unexpected class hierarchies. Such unexpected class hierarchies are the consequence of OWL's *open world assumption*: all classes can be equivalent unless declared otherwise. Whereas inconsistencies are relatively easy to detect, the cause of an unexpected class hierarchy is more difficult to root out. It is recommendable to declare classes to be disjoint whenever possible.

Whereas classes can be declared equivalent, individuals can be declared identical. Consider the identity example in Fig. 3.9, which is depicted using the '=' symbol.

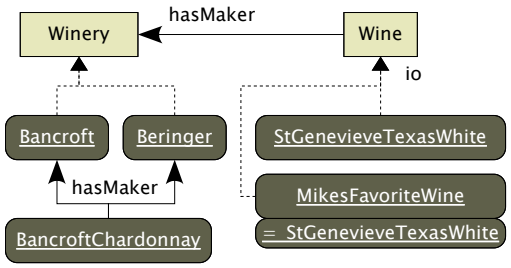

Figure 3.9: Example of identity.

"MikesFavoriteWine" is declared to be the same as "StGenevieveTexasWhite", which means that Mike likes an inexpensive local wine. Note that OWL does not have a unique name assumption. Just because two names are different does not mean they refer to different individuals. Consider the "BancroftChardonnay" wine, which has "Bancroft" and "Beringer" as its maker. Given that "hasMaker" is functional, this example is not necessarily a conflict. Unless this conflicts with other information in our ontology, it simply means that "Bancroft" = "Beringer".

Similar to disjointness for classes, it is possible to declare individuals to be *different from* another individual. This construct can be used to prevent unexpected inferences, such as the "Bancroft" winery to be the same as the "Beringer" winery.

3.7 Complex classes

In addition to simple named classes, OWL provides a number of constructs to define *complex* classes that are built out of one or more other classes. These constructs are "unionOf", "intersectionOf" and "complementOf".

Consider the example of "intersectionOf" in Fig. 3.10, which is depicted using the '⊓' symbol.

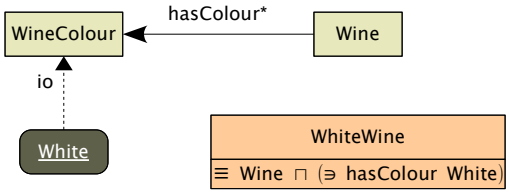

Figure 3.10: Example of "intersectionOf".

The "WhiteWine" class is defined equivalent to the intersection of the class "Wine" and the set of things that are white in colour. The members of the "WhiteWine" class are completely specified by the "intersectionOf" set operation. This means that if something is white and a "Wine", then it is an instance of "WhiteWine". Without such a definition we can only specify that white wines are wines and white, but not vice-versa.

To complement "intersectionOf", there is the "unionOf" construct. Fig. 3.11 shows an example of "unionOf", which is depicted using the '⊔' symbol.

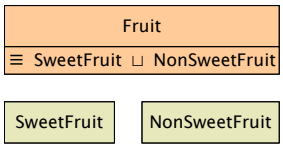

Figure 3.11: Example of "unionOf".

The "Fruit" class is defined equivalent to exactly the union of the class "SweetFruit" and "NonSweetFruit". This states that *all* fruits are sweet fruits or non-sweet fruits. It does *not* state that a "Fruit" instance cannot be an instance of both "SweetFruit" and "NonSweetFruit": that requires "SweetFruit" and "NonSweetFruit" to be declared disjoint.

In subsection 3.5, we have already given an example of "complementOf" in the property restriction "∀ locatedIn ¬WineRegion". The symbol used for "complementOf" is '¬'. The individuals defined by "¬WineRegion" are all individuals that are not wine regions. The intersection of "WineRegion" and "¬WineRegion" is typically "owl:Nothing" (empty set) and the union "owl:Thing" (everything).

Finally, OWL provides a construct a complex class built out of instances, called an *enumerated class*. Enumerated classes have a "oneOf" relationship with a closed set of individuals. This means that there can be no other instances of the enumerated class than the given individuals.

3.8 Summary

This chapter has explained what an ontology is in the context of software systems. The concrete example of the OWL ontology language has been used to illustrate what can be modelled in an ontology. We have focused on the OWL DL variant of OWL, which corresponds to description logics and can hence be supported by automated reasoning systems. For each OWL construct, we have discussed the uses and abuses, as well as the effect it has for logic inference.

The next chapter discusses how we can model platforms using OWL ontologies as well as the inferences that can be drawn from such platform models.

Chapter 4

Platform modelling

4.1 Introduction

The previous chapters discussed the background information on the Model-Driven Architecture and ontologies. The first topic represents a solution for dealing with platform diversity. The second topic provides the necessary background information to understand our approach to platform modelling. This chapter discusses how we use platform modelling as a means to deal with platform diversity.

Section 4.2 motivates the need for our platform modelling approach. It first looks at current MDA technology and shows how far it brings us in dealing with diverse platforms. Remaining issues are discussed and the need for explicit platform modelling is explained. Section 4.3 explains how the platform ontology, which forms the basis for our approach to platform modelling, is built up. Section 4.4 discusses how the platform ontology is extended for specific technical platform domains. Section 4.5 explains how platform instances, such as the Zaurus SL-C1000 PDA, are modelled. Section 4.6 discusses the definition of platform dependency constraints and how they are used. Section 4.7 discusses the limitations of our approach. Section 4.8 discusses related work and section 4.9 concludes this chapter.

4.2 Dealing with platform diversity

As has become apparent in the previous chapters, the MDA approach offers support for platform diversity up to a certain extent. The MDA aims to deal with platform diversity through automatic transformation of platform-independent software representations (PIMs) into *multiple* platform-specific software representations (PSMs). The use of platform-independent *abstractions* in our PIMs make it possible to use such PIMs across multiple platforms.

The MDA suggests to apply multiple, successive refinement transformations to the PIM in order to obtain the final PSM. This allows for reducing the complexity of each separate transformation (*divide-and-conquer*). Each of these transformations can introduce a number of platform dependencies. These platform dependencies are implicit and hardcoded in the transformations. This works fine as long as you only use such refinement transformations for the platform they were designed for. In chapter 2 we have demonstrated, however, that PIM-to-PSM refinement transformations can be made reusable over multiple platforms. The platform dependencies introduced by a single refinement transformation are minimal; it is only the chosen combination of refinement transformations that is limited to one platform – or very few platforms. It is not safe, however, to reuse a refinement transformation for another platform without knowing whether that platform satisfies the platform dependencies that the refinement transformation introduces.

That's why we propose to make platform domain knowledge explicit, as has been briefly discussed in chapter 1. This platform domain knowledge serves as a basis for the representation of platform dependencies. The representation of a platform dependency is called a *platform dependency constraint*. The explicit platform domain knowledge allows us to reason about the extent of platform dependency constraints with regard to target platform instances. If new platforms evolve, they can be added to the domain knowledge and compared against the existing platform dependency constraints. Platform dependency constraints of alternative refinement transformations can also be compared against each other, such that the most appropriate alternative may be chosen for a particular platform.

As a knowledge representation format, we have chosen OWL DL ontologies. Ontologies can serve as a common, controlled vocabulary for a domain, as has been discussed in the previous chapter. The relationships (subsumption, equivalence) between the ontology elements (classes, properties, individuals) can be used to reason about elements based on that ontology, even if those elements aren't related directly. A platform ontology allows one to base expressions about a platform on the vocabulary expressed by that ontology. By using a common model of platforms, we can reason about the relationship between a platform instance description and a platform dependency constraint, even if the two do not have a direct relationship. An example platform dependency constraint is that the Java 2 Collections framework[1] needs to be present. An example of a platform instance description is a Sharp Zaurus hand-held computer. Since both the platform dependency constraint and the platform instance description refer to the platform ontology to explain what the Java 2 Collections framework and the Zaurus hand-held computer are, one can derive whether the Zaurus hand-held computer platform satisfies the Java

[1] http://java.sun.com/j2se/1.4.2/docs/guide/collections/reference.html

2 Collections framework constraint.

OWL DL is supported by a number of automatic DL reasoners, such as Racer [MH03], Pellet [SPG+07] or Fact++ [TH06]. These automatic reasoners can be used to infer useful knowledge, such as whether a platform instance satisfies a platform dependency constraint or even whether a platform dependency constraint is *more specific* than another platform dependency constraint. *More specific* means that a platform dependency constraint forms a closer match to the platform and refers to the position of a platform dependency constraint in an OWL class hierarchy, as will be explained in detail later. The automatic inference results allow us to do automated platform dependency constraint satisfaction checks and (semi-)automated selection of model transformations, based on a platform instance description.

4.3 A platform vocabulary ontology

Before modelling any specific platform properties, a basic structure needs to be defined, into which platform extensions can be fitted. We have called this structure a platform "vocabulary ontology" in chapter 1. The word "vocabulary" refers to the fact that the domain concepts are introduced in this ontology, such that platform dependency constraints and platform instances can refer to them. Our platform vocabulary ontology is based on a *context* ontology for Ambient Intelligence that we have contributed to [PVW+04]. This context ontology includes the concept of platform. The platform is the context in which the software must run, as opposed to the context of the user. Part of the ontology that models platforms is shown in Fig. 4.1[2].

The ontology describes the following concepts:

Platform: The "Platform" class in this ontology can provide "Features", which can take the form of "Software" or "Hardware". This is denoted by the "providesFeature" property.

Feature: Features can require other features, e.g. the need for a particular "VirtualMachine" or a user interface "RenderingEngine" that supports voice communication as a "Modality". This is denoted by the "requiresFeature" property, which represents the transitive closure of required features.

Software: The "Software" class represents all kinds of software that can be present on a "Platform". It has several (disjoint) subclasses, which are intended for expressing the *direct* platform dependency constraints. Direct platform dependency constraints refer to platform elements that

[2]The full ontologies used can be found at http://ssel.vub.ac.be/ssel/research:mdd:platformkit:ontologies

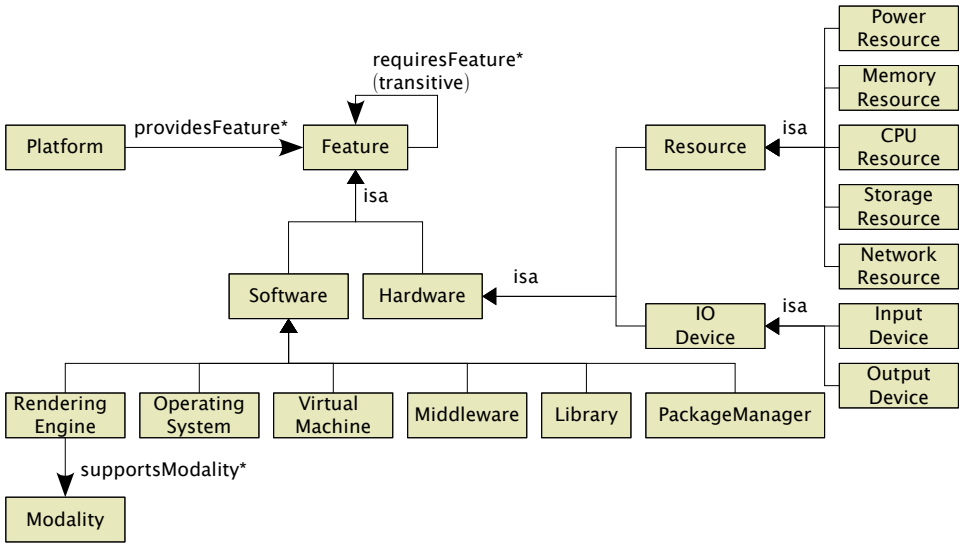

Figure 4.1: Partial view of the base platform ontology

the platform-specific software we want to deploy connects to in a direct way. They typically correspond to the "functional" requirements of the platform-specific software, such as required interfaces and expected behaviour. "Software" is disjoint from the "Hardware" class.

Hardware: The "Hardware" class represents the kinds of physical parts out of which a "Platform" is built. "Hardware" can take the form of either a "Resource" or an "IODevice" (disjoint) and is intended for expressing *indirect* platform dependency constraints. Indirect platform dependency constraints refer to the platform elements that form boundary conditions for the platform-specific software that we want to deploy to function in. They typically correspond to the "nonfunctional" requirements of the platform-specific software, such as memory and performance requirements.

RenderingEngine: The "RenderingEngine" class refers to user interface rendering software, such as Java AWT for graphical user interfaces. Rendering engines can have one or more "Modalities".

Modality: The "Modality" class refers to the various communication channels that a user interface can use, such as visual, audial or haptic.

OperatingSystem: The "OperatingSystem" class refers to the lowest-level software on a platform, that interfaces directly with the hardware. Operating systems form the first layer of abstraction on top of the hardware.

Platform-specific software is at least built on top of an operating system, if not on top of a higher-level abstraction layer.

VirtualMachine: The "VirtualMachine" class refers to a more rigorous abstraction layer than the operating system: virtual machines abstract from the hardware architecture and allow the platform-specific software to be represented in a portable binary format.

Middleware: The "Middleware" class refers to software that provides infrastructural services, such as communication and distribution. Some middleware examples are CORBA [OMG04a], DCOM [MS96] and Enterprise Java Beans [MKDS03].

Library: The "Library" class refers to any software libraries that implement interfaces on which the platform-specific software we want to deploy may depend. Subclasses of "Library" are typically not disjoint, since their classification is based on which interfaces they implement: any combination of interfaces may be implemented by a library, possibly overlapping or subsuming the functionality of other libraries.

PackageManager: The "PackageManager" class refers to the software that can install and uninstall software packages. Package managers are a vital passageway for software deployment: the platform-specific software we want to deploy must be packaged in a way that the platform understands and is able to install properly. Package managers are often bundled with the operating system, but sometimes reside in a different abstraction layer. That's why they are modelled separately.

Resource: The "Resource" class refers to the hardware that can be used by the software in a quantified manner. The platform-specific software we want to deploy may require a minimal amount of "Memory", for example. "Resource" is broken down into several (disjoint) subclasses, which reflect the typical kinds of resources in current hardware architectures.

PowerResource: The "PowerResource" class refers to the kind of power source that feeds the platform, such as mains (from wall sockets) or battery, and its capacity, if applicable.

MemoryResource: The "MemoryResource" class refers to the volatile working memory that the software can use, such as RAM.

CPUResource: The "CPUResource" class refers to the central processing unit that executes the software. In addition to its architecture (e.g. x86 or PowerPC), a CPU also has a typical performance.

StorageResource: The "StorageResource" class refers to the non-volatile storage memory that the software can use, such as a harddisk or flash memory.

NetworkResource: The "NetworkResource" class refers to any networking capacities that the software can use in terms of bandwidth and reliability. Networking resources are deliberately not modelled as I/O devices, even though network interfaces are exactly that in the technical sense: the quantifiable aspect of a network interface is more important than which exact interface is used to reach the other end.

IODevice: The "IODevice" class refers to input/output (I/O) devices that the software can use to communicate with external entities. These external entities are generally humans: the focus of an "IODevice" lies on the specific (kind of) interface that is used to communicate, rather than quantifiable aspects, such as bandwidth. I/O devices can still have quantified properties, such as screen resolution, but such properties cannot be used proportionally by the software: while software may run twice as fast on a CPU that is twice as fast, the graphical user interface does not become "twice as friendly" on a screen that is twice the size.

InputDevice: The "InputDevice" class refers to all I/O devices that can be used for data input, such as a keyboard, microphone or mouse. It is not disjoint from "OutputDevice", as some I/O devices have both input and output capabilities (e.g. a touchscreen).

OutputDevice: The "OutputDevice" class refers to all I/O devices that can be used for data output, such as a screen or loudspeaker.

The parts that have been changed from the original context ontology are the following:

- The platform-related elements have been factored out into a separate ontology.
- The "Software" and "Hardware" classes have been generalised into the "Feature" class.
- The "PackageManager" class has been added.

4.4 Extending the platform ontology

The base platform ontology described in the previous section is not detailed enough for realistic platform dependencies and platform instances, which typically refer to specific kinds of software libraries or which kinds of platforms

provide these libraries. For this purpose, the base platform ontology can be extended for particular sub-domains of platform, such as Java runtime environments (JREs). Fig. 4.2 shows part of such an ontology.

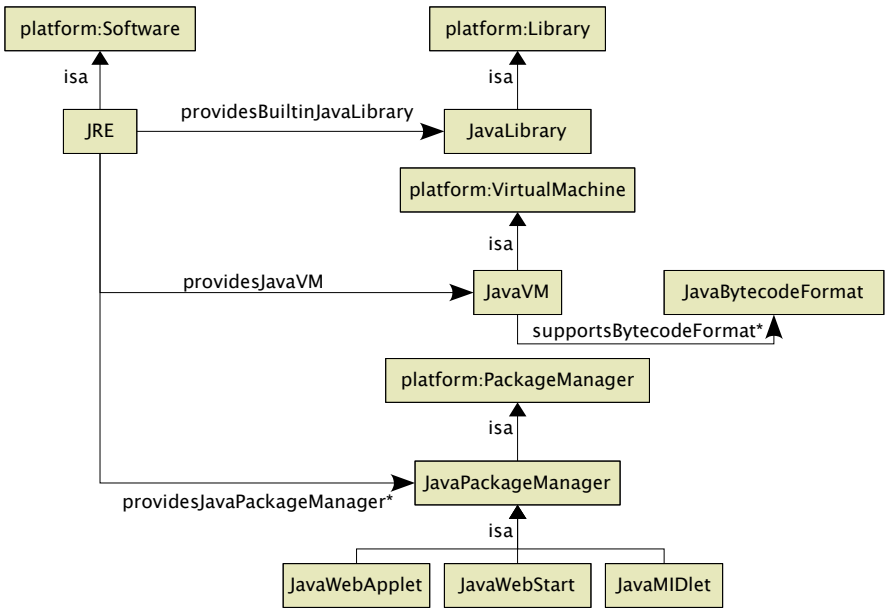

Figure 4.2: Partial view of an ontology for describing Java runtime environments

The Java ontology describes the following concepts:

JRE: The "JRE" class refers to a Java runtime environment, such as the Java Standard Edition 1.5 or J2ME Personal Profile 1.0. "JRE" is a subclass of "Software". "Software" is prefixed by "platform:" to indicate it refers to the "Software" class from the main platform ontology (see Fig. 4.1)[3]. A "JRE" consists of a virtual machine, a built-in class library and may provide a number of package managers, denoted by the "providesJavaVM", "providesBuiltinJavaLibrary" and "providesJavaPackageManager" properties.

JavaLibrary: The "JavaLibrary" class refers to a library that implements a (part of a) Java Application Programming Interface (API). It can be subclassed by libraries that implement a specific (part of a) Java API, as will be shown later.

[3]The "platform:" prefix refers to the XML namespace "platform", where the ':' serves as a namespace delimiter.

JavaVM: The "JavaVM" class refers to a Java virtual machine. A "JavaVM" can support one or more of the various existing bytecode formats ("supportsBytecodeFormat").

JavaBytecodeFormat: The "JavaBytecodeFormat" class refers to Java virtual machine bytecode formats. "JavaBytecodeFormat" has a number of instances ranging from "java1.1BytecodeFormat" up to "java1.4BytecodeFormat", "java5BytecodeFormat", "java6BytecodeFormat", "java1.1Preverified1.0BytecodeFormat" and "java1.1Preverified1.1BytecodeFormat". The last two bytecode formats refer to specialised formats for J2ME MIDP (mobile phones), whereas the other bytecode formats represent the evolutionary steps of the standard Java bytecode format.

JavaPackageManager: The "JavaPackageManager" class refers to specific parts of the Java Runtime Environment that function as a package manager: they are used to deploy and/or install Java software, such that the JRE can run it. "JavaPackageManager" currently has three subclasses: "JavaWebApplet", "JavaWebStart" and "JavaMIDlet".

JavaWebApplet: The "JavaWebApplet" class refers to the standard Java applet viewer component with web browser plugin. The applet viewer has always been part of the standard Java editions, which come with various web browser plugins. The web browser plugins allow Java applets to be run directly from the web: they are automatically "installed".

JavaWebStart: The "JavaWebStart" class refers to the Java Web Start technology that is shipped with J2SE 1.4 or up. It allows you to open a Java application descriptor file directly from the web with the JRE. The JRE then downloads all necessary components and starts the application. You can later restart the same Java application by opening the downloaded application descriptor file again. Java Web Start is meant for heavier Java applications instead of simple applets.

JavaMIDlet: The "JavaMIDlet" class refers to the Java MIDlet technology that is shipped with J2ME MIDP for mobile phones. A MIDlet is a simplified version of a standard Java applet. MIDlets come with a deployment descriptor that allows a mobile phone to download and install a MIDlet on the phone. The MIDlet can then be run from the mobile phone's menu. The "JavaMIDlet" package manager comes closest to the traditional notion of package managers, since it strongly separates the software installation from running that software.

Since the most important difference between the various JREs lies in the built-in class library from the point of view of a software engineer, we focus on the Java class library APIs. Each particular JRE complies with a particular

Java specification, such as JDK 1.1, J2SE 1.5, J2ME PP 1.0, etc. Each of these specifications have their own class library API. The differences between these APIs are significant enough to justify a separate ontology for each JRE specification. As an example, Fig. 4.3 shows part of the ontology for the J2ME Personal Profile (PP) version 1.0 specification.

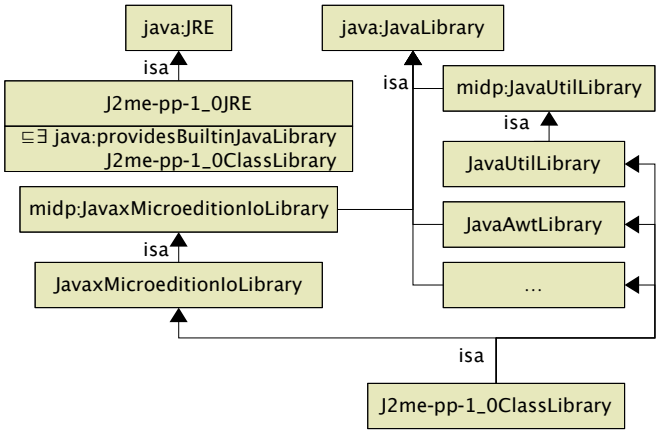

Figure 4.3: Partial view of an ontology for describing the J2ME Personal Profile 1.0 specification

The J2ME PP 1.0 ontology provides specific classes for the J2ME PP 1.0 version of the JRE and class library: "J2me-pp-1_0JRE" and "J2me-pp-1_0ClassLibrary". The "providesBuiltinJavaLibrary" property from the Java ontology is restricted such that instances of "J2me-pp-1_0JRE" must provide an instance of "J2me-pp-1_0ClassLibrary" as built-in class library. Since "providesBuiltinJavaLibrary" is a functional property, this means that there is only one built-in class library for each J2ME PP 1.0 JRE, which must be a J2ME PP 1.0 class library. All Java packages that are part of the built-in class library are represented as separate classes. The "JavaAwtLibrary" class, for example, represents the class of all Java libraries that implement the `java.awt` package of the J2ME PP 1.0 API. Since "J2me-pp-1_0ClassLibrary" represents the set of all Java libraries that implement the full J2ME PP 1.0 API, it subsumes all package library classes.

The reason that all API packages are represented as separate classes is to be able to specify overlap in the APIs of different JRE specifications. A simplified version of the "JavaxMicroeditionIoLibrary", for example, is also part of the J2ME Mobile Information Device Profile (MIDP) 1.0 API. The J2ME PP 1.0 version of "JavaxMicroeditionIoLibrary" is modelled as a subclass of "JavaxMicroeditionIoLibrary" in the J2ME MIDP 1.0 ontology, which is prefixed by "midp:". The same situation applies for "JavaUtilLibrary". Where Java API packages of different JRE versions are the same, their ontology classes are modelled as equivalent.

4.4.1 Automatic generation of Java platform ontologies

Since it is quite an error-prone and labour-intensive effort to manually determine compatibility and equivalence between different versions of Java API packages, it is better to automate this process. We chose to use model transformation to transform UML models of Java API into Java platform ontologies. For all JREs to be considered, the API, which is contained in a jar file bundled with the JRE, is reverse engineered as a UML model. It is important to accurately reverse engineer not only the standard UML class diagram elements, but also detailed UML attributes such as "visibility", "isAbstract", "isFinal", "isReadOnly" and "isLeaf", since they are necessary to properly determine compatibility. To guarantee that all required functionality is present, we use our own Jar2UML tool[4]. The Jar2UML tool is also discussed in chapter 7.

The result of this automated approach is a complete and consistent ontology representation of all Java API packages in all considered JRE versions. Any mistakes in reverse engineering, ontological modelling or the definition of compatibility and equivalence can be solved at the model transformation level. Updated ontology versions can simply be re-generated. If the updated ontology versions need to be compatible with their previous versions, for example when other ontologies depend on them, then the updated ontologies must contain at least the same classes as their previous versions (the generated ontologies contain only classes). To ensure this condition, the following technical measures must be taken:

- The reverse engineering step must generate UML models that contain at least the same packages as a previous version of the UML model.
- The model transformation step must generate at least the same ontology classes as the previous version.

That still allows us to safely vary at least the following factors:

- Contents of packages may change in the reverse engineering step.
- Packages may be added in the reverse engineering step.
- Compatibility and equivalence may be redefined in the model transformation step.
- Comparisons against other JREs may be varied in the model transformation step.

The model transformation approach used to transform the Java API UML models to platform ontologies is described in detail in appendix A.

[4] http://ssel.vub.ac.be/ssel/research:mdd:jar2uml

4.5 Platform instance specifications

Given the base platform ontology and the extensions for the relevant domains, we can model platform instances as OWL individuals. The Sharp Zaurus PDA, for example, has a J2ME Personal Profile 1.0 JRE. The ontology that describes this is shown in Fig. 4.4.

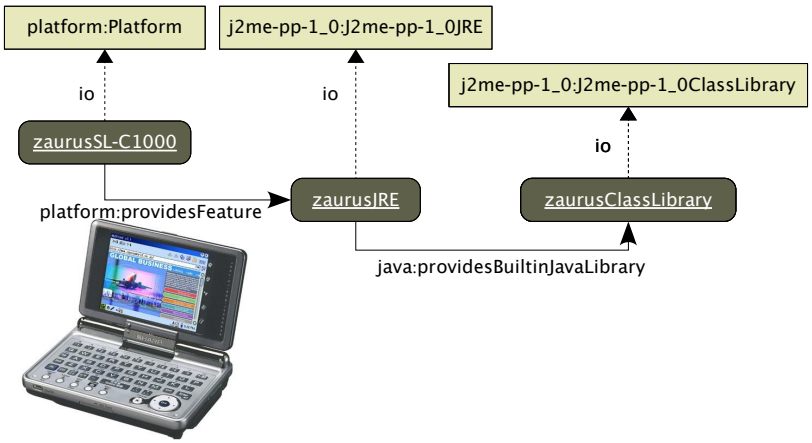

Figure 4.4: Partial platform description for the Sharp Zaurus SL-C1000 PDA

The classes "Platform", "J2me-pp-1_0JRE" and "J2me-pp-1_0ClassLibrary" are taken from the platform and J2ME PP 1.0 ontologies. The individuals, "zaurusSL-C1000", "zaurusJRE" and "zaurusClassLibrary", are instances of the "Platform", "J2me-pp-1_0JRE" and "J2me-pp-1_0ClassLibrary" classes. The "zaurusSL-C1000" platform has a "providesFeature" relationship with the "zaurusJRE" Java runtime environment. Finally, "zaurusJRE" has a "providesBuiltinJavaLibrary" relationship with "zaurusClassLibrary" to indicate that "zaurusClassLibrary" is part of the "zaurusJRE".

4.6 Platform dependency constraints

Platform dependencies can be modelled by defining new, *completely specified* classes. Such classes have *necessary-and-sufficient* constraints in addition to any *necessary* constraints. A *necessary* constraint is depicted by the *isa* relationship: whereas it is *necessary* that each instance of "J2me-pp-1_0JRE" is also an instance of "JRE", being a "JRE" instance is not *sufficient* for also being a "J2me-pp-1_0JRE" instance (see Fig. 4.3). A constraint that requires a platform with either a J2ME PP 1.0 "JavaAwtLibrary" or a Personal Java 1.1 "JavaAwtLibrary" can be defined as follows:

$$
\begin{aligned}
JavaAwtPlatform &\sqsubseteq platform : Platform \\
&\equiv \exists\, platform : providesFeature\ .\ JavaAwtJRE \\
JavaAwtJRE &\sqsubseteq java : JRE \\
&\equiv (\exists\, java : providesBuiltinJavaLibrary\ .\ pp : JavaAwtLibrary) \sqcup \\
&\quad\ (\exists\, java : providesBuiltinJavaLibrary\ .\ pj : JavaAwtLibrary)
\end{aligned}
$$

The "JavaAwtPlatform" class is defined as each platform that provides a "JavaAwtJRE". The "JavaAwtJRE" class, in turn, is defined as each JRE that provides as a built-in library either "JavaAwtLibrary" from J2ME PP 1.0 ("pp") or "JavaAwtLibrary" from Personal Java 1.1 ("pj"). The two "JavaAwtLibrary" classes represents the classes of all Java libraries that implement the `java.awt` package as specified by J2ME PP 1.0 and Personal Java 1.1.

Whenever a "JRE" instance satisfies the condition of providing a "JavaAwtLibrary", it can be classified as an instance of "JavaAwtJRE". Similarly, whenever a "Platform" instance satisfies the condition of providing a "JavaAwtJRE", it can be classified as an instance of "JavaAwtPlatform". This classification can be performed by automatic DL reasoners. This way, platform instances can be matched against a completely specified platform dependency constraint class. If an OWL individual classifies as an instance of the platform dependency constraint class, then the constraint holds for that individual. For example, the "zaurusSL-C1000" platform from Fig. 4.4 classifies as an instance of "JavaAwtPlatform", since "zaurusClassLibrary" is an instance of "J2me-pp-1_0ClassLibrary", which is a subclass of "JavaAwtLibrary" (see Fig. 4.3) and "zaurusJRE" classifies as an instance of "JavaAwtJRE".

4.6.1 Classification of platform dependency constraints

Since platform dependency constraints are represented as OWL classes, they can be classified in a subsumption hierarchy. The OWL classes at the root of the hierarchy are *least specific* and the classes at the leaves are *most specific*. In the case of platform dependency constraints, *least specific* and *most specific* refer to *platform*-specificness. A *least-specific* platform dependency constraint is the weakest constraint with regard to the platform and will satisfy the same or more platforms than the platform dependency constraints that are *more specific*. Conversely, a *most-specific* platform dependency constraint will satisfy the same or less platforms than the platform dependency constraints that are *less specific* and also forms a closer match to those platforms in terms of using as many of the provided features as possible.

The platform dependency constraints themselves are defined independently of each other and typically use *necessary-and-sufficient* conditions. Consider the following platform dependency constraints:

$$
\begin{aligned}
UtilPm &\sqsubseteq platform : Platform \\
&\equiv \exists\, platform : providesFeature\ .\ UtilJRE \\
J2UtilPm &\sqsubseteq platform : Platform \\
&\equiv \exists\, platform : providesFeature\ .\ J2UtilJRE \\
UtilJRE &\sqsubseteq java : JRE \\
&\equiv \exists\, java : providesBuiltinClassLibrary\ .\ midp : JavaUtilLibrary \\
J2UtilJRE &\sqsubseteq java : JRE \\
&\equiv \exists\, java : providesBuiltinClassLibrary\ .\ pp : JavaUtilLibrary
\end{aligned}
$$

Even though these platform dependency constraints are defined independently of each other, they are strongly related. Both are subclasses of "platform:Platform" and where "UtilPm" requires "UtilJRE", "J2UtilPm" requires "J2UtilJRE". "UtilJRE" and "J2UtilJRE" are in turn both subclasses of "java:JRE" and where "UtilJRE" requires "midp:JavaUtilLibrary", "J2UtilJRE" requires "pp:JavaUtilLibrary", which is a subclass of "midp:JavaUtilLibrary" (see Fig. 4.3). Each instance of the "pp:JavaUtilLibrary" class is also an instance of the "midp:JavaUtilLibrary" class. Hence, each instance of the "J2UtilJRE" class is also an instance of the "UtilJRE" class and each instance of "J2UtilPm" is also an instance of "UtilPm". Therefore, "J2UtilPm" can be classified as a subclass of "UtilPm". Since "J2UtilPm" is a subclass of "UtilPm", it is *more specific* than "UtilPm". All platform dependency constraints at the leaves of the OWL class hierarchy are considered *most specific*, while all platform dependency constraints at the root of this hierarchy are considered *least specific*.

An automatic reasoner for OWL-DL can automatically infer these subclass relationships. Since this can be an expensive computing task, the classification of platform dependency constraints is done only once in advance. The platform dependency constraints only need to be re-classified when they change. The platform dependency constraints change only when the models (PIMs) and/or model transformations to which they pertain have changed.

In a *configuration* of multiple refinement model transformations, several platform dependency constraints apply. When comparing different configurations to find out which is *most specific* or *least specific*, groups of OWL classes have to be compared, instead of single OWL classes. This can be done by defining an *intersection class* for each group of platform dependency constraints. An intersection class represents the intersection of the sets of instances defined by the platform dependency constraints. Hence, an instance that classifies as an instance of each of the platform dependency constraints in a group, classifies as an instance of the intersection class. The intersection classes for each group can be classified in a hierarchy in the same way that platform dependency constraints are classified. The *most specific* and *least specific* group of platform dependency constraints can now be determined.

The class hierarchy inferences made by the automatic reasoner are fed back as assertions: all new subsumption and equivalence relationships are added and subsumption relationships that have become superfluous are removed (subsumption is transitive). The OWL class hierarchy can now be used offline (i.e. without an automatic reasoner) to create a sorted list of platform dependency constraint groups. This sorted list can be *most-specific-first* or *least-specific-first*.

Since the OWL class hierarchy constitutes a partial ordering, it is possible that no subclass relationship between platform dependency constraints – or intersections of platform dependency constraints – can be inferred. In that case, it cannot be determined which (group of) platform dependency constraint(s) is more or less specific. Consider for example the "JavaAwtPlatform" and the "UtilPm" we have introduced earlier. Since the "midp:JavaUtilLibrary", the "pp:JavaAwtLibrary" and the "pj:JavaAwtLibrary" are siblings in the OWL class hierarchy, none of them is more specific than the other.

This situation is handled by providing the (groups of) platform dependency constraints as a manually sorted list, ordered by user preference. Using the inferred class hierarchy, we can now sort the list *most-specific-first* or *least-specific-first* and attempt to leave the order unchanged where no subclass relationship can be inferred. The sorting algorithm that we use for this can be found in appendix B. This process can be repeated until a satisfying order has been achieved. As the sorting process does not require the automatic DL reasoner, it can be kept fairly light-weight.

4.6.2 Satisfaction of platform dependency constraints

As soon as a platform instance description is available – usually at the time of deployment – the platform dependency constraints can be checked against this platform instance. Consider the platform instance described in Fig 4.4. The "zaurusSL-C1000" instance classifies as an instance of the "J2UtilPm" platform dependency constraint that is mentioned before: "zaurusSL-C1000" is a "platform:Platform" that provides the "zaurusClassLibrary" feature, which is an instance of "j2me-pp-1_0:J2me-pp-1_0ClassLibrary", subclass of "j2me-pp-1_0:JavaUtilLibrary". If an instance of a platform dependency constraint is found in the platform instance description, that platform dependency constraint is considered satisfied.

In the previous subsection we have used *intersection classes* for the purpose of classifying a hierarchy of *groups* of platform dependency constraints. Note that those intersection classes cannot be used for checking whether a group of platform dependency constraints is satisfied. The "platform dependency constraint" represented by an intersection class has a different meaning than "all constraints are satisfied": it defines a new platform dependency constraint, which requires that all platform dependency constraints in a group are satisfied

by a *single* instance. Therefore, a different approach is taken that considers all platform dependency constraints separately. A group of platform dependency constraints is considered satisfied if and only if each platform dependency constraint in that group is satisfied by *an* instance.

Consider the following two platform dependency constraints:

$$\begin{aligned} NetworkPm &\sqsubseteq platform : Platform \\ &\equiv \exists\, platform : providesFeature \,.\, platform : NetworkResource \\ StoragePm &\sqsubseteq platform : Platform \\ &\equiv \exists\, platform : providesFeature \,.\, platform : StorageResource \end{aligned}$$

The "NetworkPm" and "StoragePm" are both subclasses of "platform:Platform". As such, they **can** be satisfied by the same platform instance. If we target *two* platforms for our software, of which one is networked and another provides the disk storage, then both platform dependency constraints are meant to be satisfied by different platform instances. Distributed systems typically target more than one platform.

The procedure for checking platform dependency constraint satisfaction uses the sorted list that results from the classification of platform dependency constraints. Each list entry corresponds to a configuration of model transformations, each of which have platform dependency constraints. The list entries that have at least one unsatisfied platform dependency constraint are removed from that list, such that the resulting list contains only entries with all platform dependency constraints satisfied. A configuration of model transformations can now be automatically chosen by picking the first entry from the pruned list. Depending on how the list was sorted, this entry is *most specific* or *least specific*.

4.7 Limitations

Our platform modelling approach currently has some known limitations. This section discusses these limitations and suggest how these limitations can be mitigated.

4.7.1 Constraint interaction

Platform dependency constraints may have an influence on each other. Consider the "JavaAwtPlatform" and "UtilPm" platform dependency constraints given earlier in this section: one requires a JRE that provides the `java.awt` library and the other requires a JRE that provides the `java.util` library. There

happen to be platforms that provide multiple JREs, however. The "zaurus-SL-C1000" PDA, for example, can have any combination of J2ME Personal Profile 1.0, Personal Java 1.1 and J2SE 1.3 installed on it. JREs cannot useeach other'ss class libraries, however: these class libraries are integrated monolithic components that also contain integrated native code.

If the platform dependency constraints mentioned before refer to a simple Java applet, then that applet is meant to run within a single JRE. An additional (meta-)constraint that applies here, is that both platform dependency constraints must be satisfied by the *same* JRE. If the platform dependency constraints refer to a distributed Java application, then it may be just fine that the constraints are satisfied by different JREs or even different platforms. It is currently not possible to express such "derived" platform dependency constraints. A platform dependency constraint that requires all *applicable* JRE constraints to be satisfied must first know which JRE constraints are in fact *applicable*.

This problem is mitigated by the fact that platform instance descriptions can be limited to a single platform/JRE. It is also mitigated by the fact that a limited amount of model transformation configurations are used: (1) there will be no configurations that don't run on any platform and (2) there will be no configurations that are sub-optimal for all platforms (e.g. through the use of a primitive `java.util` library in combination with the `javax.swing` library, where all JREs that support `javax.swing` include an advanced version of `java.util`).

4.7.2 Performance of determining constraint satisfaction

While the class hierarchy of platform dependency constraints can be determined beforehand and is done only once for a given set of model transformations, constraint satisfaction must be determined each time a platform instance description becomes available. In other words: each time the software is deployed on a platform instance, the platform dependency constraints of each configuration must be checked. Since constraint satisfaction checking requires the use of an automatic DL reasoner, we are bound by the computational complexity of the reasoning process. OWL DL can be translated to the $\mathcal{SHOIN}(\mathbf{D})$ description logic in polynomial time [HPS04], but determining $\mathcal{SHOIN}(\mathbf{D})$ satisfiability has a complexity of NExpTime [Tob01].

Our use of OWL DL in the platform ontologies so far has remained mostly limited to OWL Lite, however: OWL Lite can be translated to the $\mathcal{SHIF}(\mathbf{D})$ description logic, which lacks the "hasValue" property restriction and enumerated classes ($\mathcal{O}$), and is limited to functional properties ($\mathcal{F}$) instead of cardinality constraints with arbitrary numbers ($\mathcal{N}$). The complexity of determining

$\mathcal{SHIF}(\mathbf{D})$ satisfiability still has a complexity of ExpTime, however [Tob01]. Our platform ontologies are *mostly* limited to OWL Lite, because OWL Lite places further restrictions on the OWL language, which go beyond what is necessary for $\mathcal{SHIF}(\mathbf{D})$: class disjointness, intersection, union and complement are not allowed in OWL Lite. Our usage of OWL DL is further limited, in that we also have not used any inverse properties in our platform ontologies ($\mathcal{I}$), reducing our DL usage to $\mathcal{SHF}(\mathbf{D})$.

This translates to real-world values of up to 15 seconds response time when determining constraint satisfaction for six different configurations of our instant messenger case study against a description of a standard JDK 1.6 PC. This performance is achieved on a dual Intel Core2 Quad Xeon CPU at 2.33 GHz with 8 GB of RAM[5]. The entire knowledge base of ontologies that is used in our case study contains approximately 1000 concepts, of which half is completely defined using an equivalence relationship. Since our case study still represents a fairly simple situation, performance is expected to degrade for real-world cases. It should be noted, however, that the bulk of ontology data will always be made up of the platform vocabulary ontologies, not the platform dependency constraints or the platform instance descriptions (there are only 21 completely defined platform dependency constraint concepts for the instant messenger case study and 6 individuals in the JDK 1.6 PC platform description).

If we want to achieve PSpace complexity, however, we certainly need to remove transitive properties, as $\mathcal{ALC}_{trans}$ already has a complexity order of ExpTime [BCM+03]. Our platform ontologies currently use exactly one transitive property, "requiresFeature", which is not used for platform dependency constraints. In addition, the platform ontologies use role hierarchy at two occasions: "requiresFeature" has a subproperty named "directlyRequiresFeature", which represents the non-transitive variant of "requiresFeature", and "providesFeature" has two subproperties named "providesSoftware" and "providesHardware", which only exist for historical reasons (both properties have been refactored into one and the subproperty relationship provides backward compatibility). If we choose to eliminate those, our usage of OWL is reduced to $\mathcal{ALCF}(D)$. A superset of $\mathcal{ALCF}$, $\mathcal{ALCQ}$, has already been proven to have PSpace complexity [Tob01].

A practical consideration to keep in mind, however, is that currently available DL reasoners use an algorithm that is designed for use with a standard description logic, such as $\mathcal{SHIQ}$, $\mathcal{SHIF}$ or $\mathcal{SHOIN}$, which is limited to the accompanying complexity order. On top of that, different DL reasoners have different levels of optimisation (within the complexity order limits): RacerPro is a commercial reasoner that typically performs better than Pellet and Fact++, both of which are similar in performance. It is not clear how these reasoners perform on a strictly limited DL, like $\mathcal{ALCF}$, versus sporadic usage

[5]Current DL reasoners do not take advantage of multiple CPUs, however.

of transitive roles and role hierarchies in a standard $\mathcal{SHIF}$ DL.

Finally, the whole tool chain plays a role in the performance of determining platform dependency constraint satisfaction: the ontology storage layer (for reading and updating ontologies), the reasoner interface (native or via DIG[6]) and the reasoner itself.

4.8 Related work

The lack of explicit platform models is also discussed in [ADvSP04]. They introduce *abstract platforms*, which describe a set of elements to model a PIM against. This set of elements includes design artefacts that are available in a target platform (classes, interfaces) and design constructs that can be mapped to that platform (stereotypes, profiles), e.g. with model transformations. The goal of abstract platforms is to ease platform-independent modelling. Our platform models can be used to define explicit platform dependencies for each abstract platform.

In [TBA04], platform selection rules are discussed, which allow for pre-selecting a number of target platforms. In that way, less platforms need to be supported. In our case, platform selection rules can be used to narrow down the amount of platform domain concepts (e.g. Java virtual machines) that need to be modelled to support the pre-selected target platforms.

We use OWL ontologies to represent our platform models. In [MRZ+06], an ontology-based software comprehension framework is presented. Our platform ontologies can contribute to this framework and to the knowledge management of software development in general. Our use of ontologies to support the MDA is also in line with a trend where model-driven approaches increasingly converge with ontology-based approaches [RB06][KKK+06].

In [BDD+05], an infrastructure for combining UML/MOF models and ontologies is introduced. Such an infrastructure can be useful for a better integration of platform dependency constraints into configuration languages that are based on MOF.

In [OMG06c], the OMG proposes a standardised meta-model for ontologies. We have in fact used an EMF-based version of this meta-model[7] to transform UML models of the API of specific Java platforms, such as J2ME PP 1.0 and J2ME MIDP 1.0, to ontologies of these platforms using the ATLAS transformation language. The UML models of the API are in turn generated using the Jar2UML tool[8].

[6] http://dl.kr.org/dig/interface.html
[7] http://www.alphaworks.ibm.com/tech/semanticstk
[8] http://ssel.vub.ac.be/ssel/research:mdd:jar2uml

4.9 Summary

This chapter has explained the shortcomings of MDA in dealing with platform dependencies. While the technical infrastructure for multiple, alternative model refinement transformations exists, there is no method to safely and efficiently manage the application of model transformations. That's why model transformations are currently only used for a single, tested platform. If we can reason about the platform dependencies that each model transformation introduces, then we can find out the exact set of platforms for which a particular model transformation is valid.

We have stated that we need to make platform domain knowledge explicit in order to reason about platform dependencies of model refinement transformations – and software artifacts in general. Our platform models are expressed as OWL DL ontologies, as ontologies are well-aligned for expressing domain knowledge. We have demonstrated that our OWL DL platform ontologies are loosely coupled: platform dependency constraints and platform instance descriptions do not refer directly to each other. Instead, both platform dependencies and platform instances are described in terms of a common vocabulary ontology. We have also demonstrated the extensibility of the platform vocabulary ontology for the domain of Java Runtime Environments (JREs). Model transformation, which is a central technology in MDA, has proven useful for the automatic generation of the JRE platform ontologies.

OWL DL is supported by a number of tools, amongst which are automated DL reasoners. These reasoners can be used to determine platform dependency constraint satisfactions as well as to classify a hierarchy of platform dependencies. Such a hierarchy can be used to determine the *most-specific* or *least-specific* platform dependency constraint. Platform dependency constraints are linked to software artifacts, so we can use the hierarchy information to (semi-)automatically select the most appropriate software artifact for a given platform.

Both platform dependency constraint satisfaction checks and platform dependency constraint hierarchy classification can be done for individual platform dependency constraints as well as groups of platform dependency constraints. This allows us to support software artifacts with multiple platform dependency constraints, as well as *configurations* of multiple software artifacts, as typically occurs in the case of model transformations.

Two known limitations of our platform modelling approach have been discussed: platform dependency constraint interaction and automated reasoner performance for constraint satisfaction checks. Finally, this chapter has shown where our work is situated with respect to related work in the area of MDA and ontologies.

The following chapter introduces Software Product Lines (SPLs). SPLs provide a number of technologies to deal with configuration, a missing link in

the MDA. SPLs can contribute in this way and a number of other ways to the MDA, as has been explained in chapter 1.

Chapter 5

Software Product Lines

5.1 Introduction

A Software Product Line (SPL) [CN01] refers to a set of software (or software-intensive) systems that share a number of common features and have a similar purpose. The term was coined by the Software Engineering Institute (SEI), which has acted as a guardian group to track and promote all SPL-related research and technology. The SEI provides the following definition for a SPL:

> "A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN01]

According to this definition, any software that is developed using the MDA approach can be considered as an SPL, since each platform-specific software product shares its features with other platform-specific versions of that software product. This shared functionality is typically specified in the PIM. Whenever the MDA and SPL technology are used in combination, we will speak of *MDA-based SPLs*.

The words *core asset* and *feature* are named as building blocks for a SPL. Core assets and features are defined as follows:

> "*Core assets* are those assets that form the basis for the software product line. Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is key among the collection of core assets." [CN01]

> "A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between systems." [CHE04]

The development of a new SPL generally starts off with the analysis of common parts and varying parts in the software variants: the so-called Commonality/Variability Analysis (CVA) of software features [CHW98]. The result of such an analysis can then be recorded in a *feature model* [CHE04] describing whether features are *mandatory*, *optional* or *alternatives*. The feature model provides the basis for the *configuration rules* of a software variant in this way. Configuration can be done using a *domain-specific language* (DSL) [DK02][CHE05b] that is derived from the feature model.

The remainder of this chapter discusses the practises of Commonality and Variability Analysis, Feature modelling and Configuration in more detail.

5.2 Commonality and Variability Analysis

Commonality and Variability Analysis (CVA), or *Scope, Commonality and Variability* (SCV) analysis, provides a framework for software engineers to think about the software product line they are developing. When software engineers know that a number of software artefacts will have many similarities and is likely to evolve as a group as well, they can set the initial *scope* for a SPL. Then, through systematic analysis of *commonalities* and *variabilities* of software features within this group, this scope can be justified and/or adapted.

In the Family-Oriented Abstraction, Specification and Translation (FAST) approach, commonality and variability are formally defined in terms of sets:

> "A commonality is an assumption held uniformly across a given set of objects (S). Frequently, such assumptions are attributes with the same values for all elements of S. Conversely, a variability is an assumption true of only some elements of S, or an attribute with different values for at least two elements of S." [CHW98]

Consider an SPL set "InstantMessagingClient", consisting of the member elements "JabberClient", "ICQClient" and "SMSClient". An example commonality is that all "InstantMessagingClient" members can send and receive messages. An example variability is that only "JabberClient" and "ICQClient" can store contacts on the messaging server. "SMSClient" has to keep its own local contact list.

CVA can be applied hierarchically: any subsets formed by the discovery of a variability can be further divided into subsets. For example, the variability of storing contacts on the server splits the set "InstantMessagingClient" into the subsets "JabberICQClient" and "SMSClient". A variability that distinguishes "JabberClient" from "ICQClient" is the communication protocol: "JabberClient" uses a protocol named "XMPP", whereas "ICQClient" uses the "Oscar" protocol.

FAST uses five main steps for CVA:

1. Establish the scope: the collection of objects under consideration.
2. Identify the commonalities and variabilities.
3. Bound the variabilities by placing specific limits on each variability.
4. Exploit the commonalities.
5. Accommodate the variabilities.

Once the commonalities and variabilities are identified, the variabilities must be limited in order to have a clear scope for the software product line. Our example variability of communication protocol is limited to the values "XMPP", "Oscar" and "SMS". Other instant messaging protocols, such as "MSN" or "GoogleTalk" are not considered within the current (initial) scope of the product line. Of course, future expansion of the product line scope may well add extra communication protocols to support user needs.

When the scope of the product line has been lined out, *core assets* should be developed for each commonality. One of the most important core assets of a software product line is its *architecture*. The definition of *architecture* given in the previous chapter holds for SPLs as well: an SPL architecture specifies parts and connectors and sets the rules for the interaction of the parts via the

connectors. The CIM and the PIM in an MDA-based SPL form the product line's architecture. All products in the product line – PSMs in the case of MDA – have to follow that architecture.

A very important aspect of the product line architecture design is that it must accommodate the variabilities of the product line. If we consider the PIM of the instant messenger product line (see Fig. 2.4, chapter 2), it contains an abstract "Network" class with a static "discoverNetworks" operation. This design choice accommodates the variability of having multiple network protocols: new network protocols can be defined by subclassing the "Network" class and implementing its abstract operations. Since we placed a limit on the multiple network protocol variability, it is possible to define a simple behaviour for the "discoverNetworks" operation, since it only has to look for the presence of a predefined set of network protocols. An increase of the product line scope requires an update of its architecture in this case.

Variabilities are generally implemented using *optional* features, which may or may not be included in a particular product in the product line. Fig. 5.1 shows the optional "Jabber" feature of the instant messenger product line.

The "Jabber" feature builds on top of the JabberWookie component, which is a third party open source library. The "Jabber" class connects to the Jabber network server via the "Client2Server" class provided by JabberWookie. In addition, the "Jabber" class implements several predefined JabberWookie interfaces, such that it can receive special network messages. The "Jabber" class includes three nested classes – indicated by the "From Jabber" comment – to keep track of its connection state and adapt its behaviour accordingly. Any information sent to the network must be sent asynchronously, since network activity is not meant to block the main application logic. Finally, the "Jabber" class is an abstract class, having an abstract "connect" operation. This is because the "Jabber" feature is designed to work on an abstract input and output stream instead of an actual network socket. The Java implementation of sockets is unfortunately not common for all Java platforms, so we've identified an additional variability within the "Jabber" feature. There are two alternative Jabber network transport subfeatures that either use the J2ME MIDP networking interface or the default `java.net.Socket` interface: "MEJabberTransport" and "DefaultJabberTransport".

The latter example shows that CVA is an iterative activity: during the accommodation of the "Jabber" feature, it became apparent that different implementations of the network transport layer were necessary. Hence, another variability was identified.

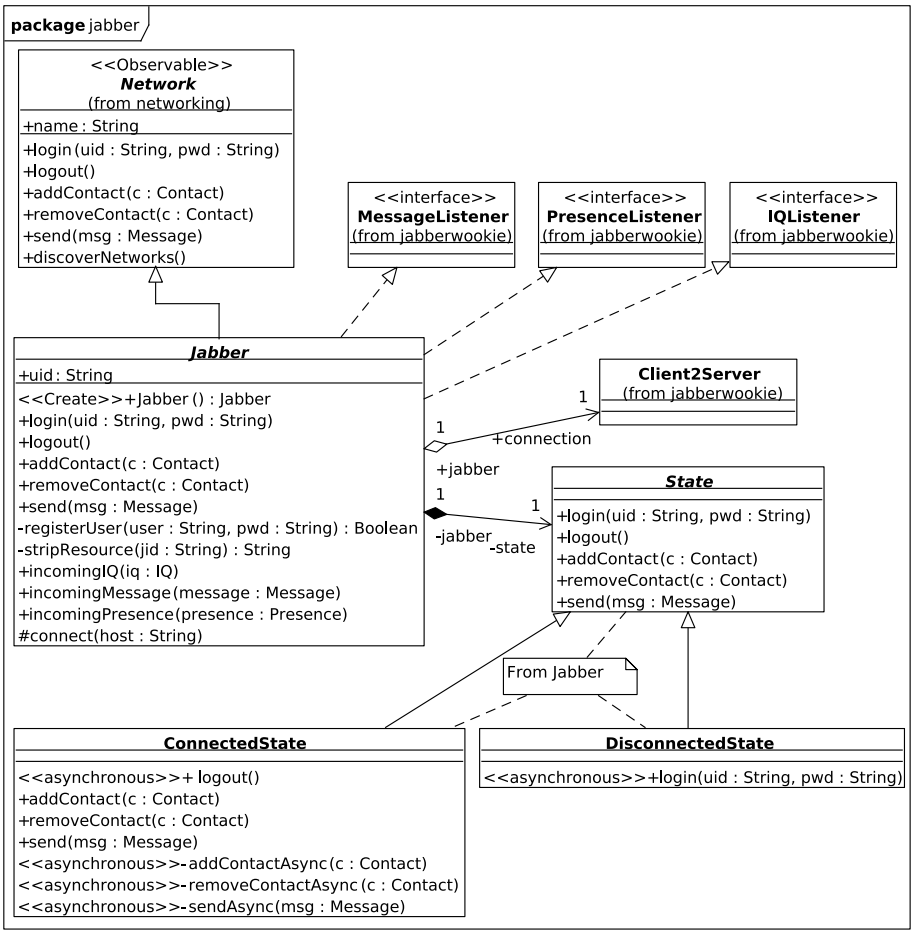

Figure 5.1: A UML Class diagram showing part of the instant messenger model for the Jabber feature.

5.3 Feature modelling

Commonalities and variabilities of a software product line can be recorded in a *feature model*. Feature models specify these commonalities and variabilities in terms of – common and variable – *features*. Feature models were introduced by Kang et al. as part of the Feature Oriented Domain Analysis (FODA) method [KCH+90]. They were later adopted and enhanced for use in Generative Programming [CE00]. Several enhancements of feature models have been proposed over time and have been consolidated by Czarnecki et al. in [CHE05b]. A formalisation of feature models is described in [CHE05a]. Fig. 5.2 shows the feature model for the instant messenger product line example.

The root element of the feature model is called the *concept*. The concept represents the software product line for which we want to describe the features. Our example feature model describes the concept “InstantMessenger”. All

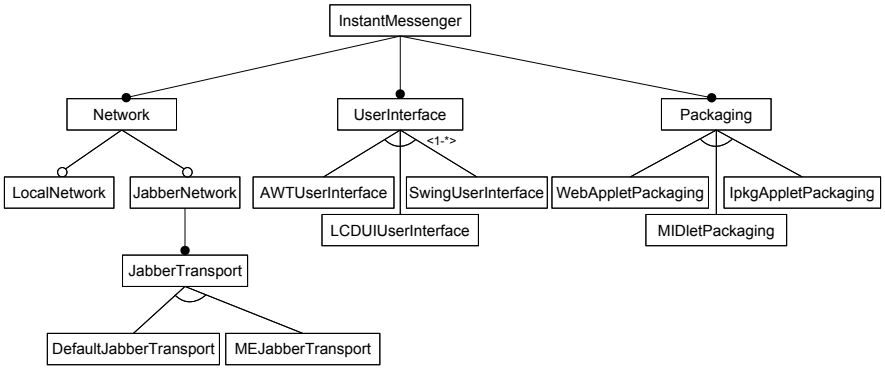

Figure 5.2: The feature model of the instant messenger product line.

other elements ("Network", "UserInterface", etc.) represent the features and are directly or indirectly connected with lines to the root concept. An open circle at the end of a line designates an *optional* feature, whereas a closed circle represents a *mandatory* feature. "JabberNetwork" and "LocalNetwork" are optional features, whereas "Network", "UserInterface" and "JabberTransport" and "Packaging" are mandatory features.

Each feature can have *subfeatures*: when including a feature, one also has to consider its subfeatures. If we choose to include the "JabberNetwork" feature, we also have to include its mandatory "JabberTransport" subfeature. "JabberTransport" has two *alternative* subfeatures: "DefaultJabberTransport" and "MEJabberTransport". Exactly one of these features must be included. It is also possible to specify cardinality constraints for a set of alternative features: we must include at least one of the features "AWTUserInterface", "SwingUserInterface" and "LCDUIUserInterface", for example, which is indicated by the $\langle 1 - * \rangle$ group cardinality constraint. Note that the $*$ notation is not officially supported by Czarnecki's feature diagram notation:

> "A feature group expresses a choice over the grouped features in the group. This choice is restricted by the *group cardinality* $\langle n - n' \rangle$, which specifies that one has to select at least n and at most n' distinct grouped features in the group. Given that $k > 0$ is the number of grouped features, we assume that the following invariant on group cardinalities holds: $0 \leq n \leq n \leq k$." [CHE05b]

According to this definition, we should use $\langle 1 - 3 \rangle$ as a group cardinality instead, as there are three grouped features to choose from. Doing this, however, creates problems when adding new features to the group: the cardinality constraint must be reviewed and/or updated accordingly. But what was the

reason for the cardinality constraint's upper bound again? Was it set to 3, because there should never be more than three user interfaces in any product? Or was it set to 3, simply because there were only three user interface features available at that time? Introducing the $*$ notation to group cardinality constraints takes away this ambiguity and properly models our original "at least one" intention.

In addition to cardinality constraints on groups of alternative features, it is also possible to set explicit cardinality constraints on features themselves. The open circle and closed circle notation to represent optional and mandatory features, for example, can also be expressed via the $[0-1]$ and $[1-1]$ cardinality constraints. Normally, each feature is a *singleton*: a feature is either included or not, and is instantiated only once. For some features, however, it makes sense to include them more than once. A car can include 3 or 4 wheels, for example, where the wheels are all the same (i.e., there are no subfeatures of the "wheel" feature). This situation is modelled by writing a $[3-4]$ cardinality constraint next to the "wheel" feature. Note that the $\langle\rangle$ and [] characters have nothing to do with mathematical intervals: they serve only to distinguish between group cardinalities and feature cardinalities.

Features can also share subfeatures. This is useful when two or more feature have an identical tree of subfeatures. Consider another subfeature of "Network" in Fig. 5.3, called "GabberNetwork", which is compatible with "JabberNetwork". Both "GabberNetwork" and "JabberNetwork" have a mandatory "JabberTransport" subfeature, including all subfeatures of "JabberTransport". It is not necessary to model the "JabberTransport" feature for each of its superfeatures. Instead, a curved, dashed line connects the "JabberTransport" feature to the closed circles that indicate that "JabberTransport" is mandatory for both "JabberNetwork" and "GabberNetwork".

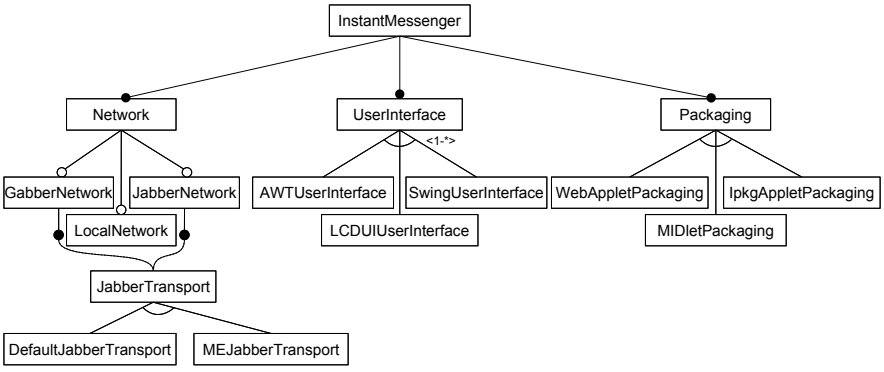

Figure 5.3: The feature model of the instant messenger product line.

When selecting features, the choice of subfeatures is always localised to its super-feature. This means that for shared subfeatures, feature selection is done

separately for each time that subfeature is referenced. In our example, it is possible to select "DefaultJabberTransport" as a subfeature of "GabberNetwork" and select "MEJabberTransport" as a subfeature of "JabberNetwork".

5.3.1 Automated analysis

There are some specific cases in which the standard feature modelling approach is not sufficient to describe the composition rules for the features. These cases are typically caused by *feature interaction* constraints [RMR05]. Standard feature models can deal with feature interaction constraints as long those constraints follow the hierarchy of the feature model. As soon as feature interaction constraints cut across this hierarchy, feature models can no longer express those constraints.

Assume that the "IpkgAppletPackaging" feature requires that the overall size of the installable package can be at most 1 MB. This means that only a limited number of all features that take up space can be chosen. Apart from the extra information required to determine whether the constraint is satisfied – we need to know the amount of packaging space each feature takes up – we must be able to describe the feature limit regardless of where the features are in the feature model hierarchy. In addition, this size limit constraint *only* applies if the "IpkAppletPackaging" feature is chosen.

Another problem is the *consistency* of feature interaction constraints. Assume for example that the "WebAppletPackaging" feature requires the "SwingUserInterface" feature. If another constraint states that the "SwingUserInterface" excludes the "WebAppletPackaging" feature, then we cannot select the "WebAppletPackaging" feature without violating either one of these constraints: the constraints are considered to be inconsistent.

There exists various research on the automated analysis of feature models, which typically describe the feature model in a formal language domain [BBRC06]. The formal language domain that is used typically improves the expressiveness of feature interaction constraints. Automated analysis becomes a must when the number and complexity of feature interaction constraints goes up. Amongst other things, automated analysis can verify consistency of feature interaction constraints as well as the satisfaction of all feature interaction constraints for a particular configuration of features (see also next section). Benavides et al. have made a survey of automated analysis approaches and their usage scenarios in [BRCTS06]. The surveyed approaches are based on the following language domains:

Propositional logic: Several approaches propose to translate basic feature models into propositional formulas, which allows for the use of existing solvers [Man02][ZZM04][SZW05].

Description logic: There is one approach listed that translates feature models to OWL DL, which corresponds with description logic [WLS+07]. It should be noted that there is at least one other approach to feature modelling that uses description logic [VB01].

Constraint programming: The author of the survey proposes to translate feature models to a Constraint Satisfaction Problem (CSP), which can be solved by automatic constraint solvers[BTRC05].

Ad-hoc: The survey lists a number of automated approaches that do not use formal language underpinnings. As such, the survey assumes their expressiveness to be the same as basic feature models.

5.4 Configuration

There are several methods for configuring a product in a software product line, such as staged configuration of feature models [CHE05b], Stepwise Refinement [BSR04] as well as via Domain-Specific Languages (DSLs) [DK02]. Each of these methods can be supported by automated analysis (see previous section). Van Deursen and Klint have studied the relationship between feature modelling and domain-specific languages in [DK02]. They defined a grammar for their Feature Description Language (FDL) to give rigorous structure to a feature model. It is later noted that the grammar for this language lies very close to the grammar of the grammar definition language itself, making it possible to define the features themselves as first-class language elements. Finally, they present a UML class diagram to illustrate how a feature model can be implemented. UML class diagrams are in turn very close to MOF models, which are commonly used in MDA to describe meta-models. One can now extrapolate and see how it is possible to translate the feature model to a MOF meta-model. Such a meta-model effectively describes a Domain-Specific Modelling Language (DSML) [LBM+01][TR03]. Any model written in this DSML represents the configuration of a software variant.

Models play an increasingly important role in SPL frameworks, where products are generated from core assets with the help of Product Models [Kru06]. These Product Models can be feature models or they can be written in a DSML. Whereas the definition of a textual DSL is typically based on a grammar, the definition of a DSML uses a meta-model to describe its (abstract) syntax.

While all of the listed configuration methods are capable of SPL configuration, only feature models and DSMLs provide sufficient mechanisms for *bounded combinatorics* [Kru06]. Bounded combinatorics is a methodology for constraining, eliminating or avoiding combinatoric complexity in SPLs. It is

based on the premise that only a small subset of all correct product configurations is actually useful and deals with the problem that only a limited set of product configurations can be maintained. The configuration method needs to provide two mechanisms to support bounded combinatorics:

- Modularity and encapsulation
- Composition and hierarchy

For feature models, modularity and encapsulation refer to the partitioning of a feature model. That partition can then be used across multiple (parts of) a product line and can be imported in more specialised feature models. DSML meta-models can be partitioned in a similar way. Composition and hierarchy, in the context of feature models, refer to the subfeature mechanism, where subfeatures only become relevant when their superfeature is selected. Composition and hierarchy in DSMLs are implemented through containment relationships between meta-classes.

DSMLs currently have better tool support than feature models and can be used on a larger scale because of their (optimised) tool support. DSMLs also align well with the concepts and tools from MDA and do not create any technology overhead for SPLs that already use the MDA. Hence, we will focus on configuring product lines using DSMLs from here on. The following subsections discuss the various aspects of product line configuration with a DSML: a configuration language meta-model, product configuration models and configuration model transformations that translate a product configuration to an executable specification.

5.4.1 Configuration language meta-model

The meta-model for the DSML can be derived from a feature model in the same way that Van Deursen and Klint derive a UML class diagram in [DK02]. Since the purpose of a feature model is different from the purpose of a configuration DSML, the feature model is not simply translated into a meta-model. A feature model primarily serves to communicate the results of the CVA, whereas the DSML is only meant to describe individual configurations in a product line. The features from the feature model that are not relevant for configuration purposes, such as the mandatory features, are therefore omitted from the DSML meta-model. This is a form of staged configuration [CHE05b]. Fig. 5.4 shows the meta-model for the configuration of the instant messenger product line example.

A feature hierarchy in a feature model is translated to meta-classes with containment relationships. The root concept “InstantMessenger” (see Fig. 5.2) is translated to an “InstantMessengerConfiguration” meta-class with several outgoing containment relationships. The mandatory “Network” feature is not

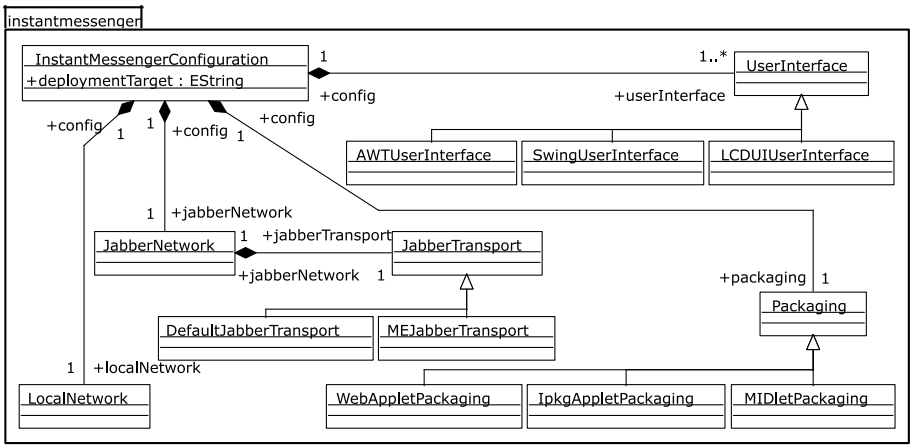

Figure 5.4: The meta-model of the instant messenger configuration language.

modelled, since it is always included in a configuration. Its optional subfeatures "LocalNetwork" and "JabberNetwork" *are* modelled, since they are optional and represent a configuration choice. In the case of alternative features, an additional, abstract meta-class is used to represent the group of alternatives. In our example feature model, all groups of alternative features are already represented by a mandatory superfeature: "JabberTransport", "UserInterface" and "Packaging". Each of the alternatives is modelled as a subclass of the group meta-class. "JabberTransport", for example, has two alternatives: "DefaultJabberTransport" and "MEJabberTransport". The containment relationship between "JabberNetwork" and "JabberTransport" now guarantees that exactly one of the alternative subclasses of "JabberTransport" must be chosen. In the case of alternatives with a cardinality constraint other than $\langle 1-1 \rangle$, a multiplicity is defined on the corresponding containment relationship in the meta-model. The $\langle 1-3 \rangle$ cardinality constraint of the "UserInterface" subfeatures, for example, is translated to a "1..*" multiplicity on the containment relationship between "InstantMessengerConfiguration" and "UserInterface".

There is a small mismatch between feature models and meta-models: all features in a feature model are singletons unless an explicit cardinality constraint on that feature specifies otherwise. Meta-models cannot express cardinality constraints on meta-classes and meta-classes can be instantiated any number of times. This mismatch is demonstrated by the subclasses of the "UserInterface" meta-class: it is possible to include multiple instances of "AWTUserInterface" in an "InstantMessengerConfiguration". This mismatch can be solved by the addition of an OCL invariant:

```
context InstantMessengerConfiguration inv:
  userInterface->select(oclIsKindOf(AWTUserInterface))->size() = 1 and
  userInterface->select(oclIsKindOf(SwingUserInterface))->size() = 1 and
  userInterface->select(oclIsKindOf(LCDUI2UserInterface))->size() = 1
```

OCL support exists for some meta-modelling frameworks. EMF, for exam-

ple, provides the EMF Validation component[1]. EMF has been explained in chapter 2.

The results of the automated analysis techniques for feature models can be applied to the configuration language. Since meta-classes in the meta-model correspond to feature selections in the original feature model, a translation can be made to the formal language domain of a particular analysis technique. Complex feature interaction constraints used in the automated feature model analysis may be translated to OCL constraints on the configuration language meta-model.

5.4.2 Configuration models

A product configuration model conforms to the configuration language meta-model for the product line. Fig. 5.5 shows an example configuration model for the instant messenger product line, displayed inside the EMF-based (see chapter 2) configuration model editor.

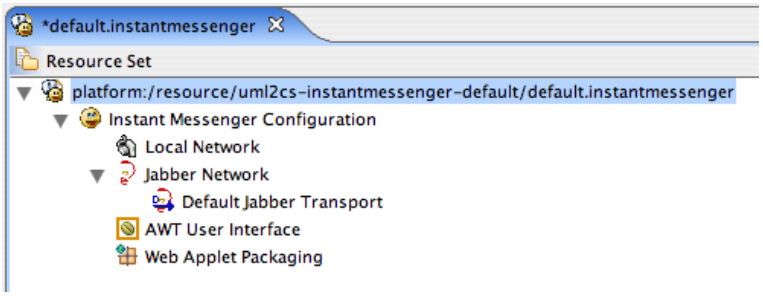

Figure 5.5: An example instant messenger configuration model as displayed by the EMF model editor.

The configuration model is displayed as a tree, where the root node represents the configuration model itself. Each sub-node represents an instance of a meta-class. In addition to the text labels on each node, icons are used to indicate which meta-class each node instantiates.

The EMF-based editor includes validation functionality that checks the conformance of each model to its meta-model. Hence, each model validated by this editor is guaranteed to follow the rules that are set by its meta-model. This validation functionality also includes a check of any OCL invariants that have been defined for the meta-model.

5.4.3 Configuration transformation

DSMLs use one or more model transformations to describe its semantics in terms of another language domain. That other language domain typically con-

[1] http://www.eclipse.org/modeling/emf/?project=validation

cerns an executable language. Product configurations are expressed through configuration models that conform to the configuration language meta-model for the SPL. A model transformation for the configuration language can transform a configuration model into an executable program that performs the specified configuration.

In the case of our example instant messenger product line, this model transformation is written in ATL (see chapter 2, subsection 2.4.2) and generates an Ant "build.xml" script from a configuration model. Ant scripts are the Makefile equivalent for Java-based software development: they execute the different steps required for building and/or packaging the software. When this Ant script is executed, it will generate the code for the instant messenger product according to the configuration model. Below is an excerpt of this model transformation:

```
helper def : ModelPath : String = '/uml2cs-instantmessenger-model/models';

rule ConfigRoot {
  from s : CFG!"instantmessenger::InstantMessengerConfiguration"
  -- <project name="s.buildPath" default="all">
  to root : XML!"XML::Root" mapsTo s (
    name <- thisModule.ConfigHeader(s)),
     default : XML!"XML::Attribute" (
      parent <- root,
      name <- 'default',
    value <- 'all'),
  -- <target name = 'all' depends = '...'>
     allTarget : XML!"XML::Element" (
      name <- 'target',
    children <- thisModule.AllTarget(s),
    parent <- root),
  -- <target name = 'InstantMessengerModel' depends = 'allBase'>
     imTarget : XML!"XML::Element" (
      name <- 'target',
    children <- thisModule.ModelTarget(
      'InstantMessengerModel',
      'inModel',
      'im',
      thisModule.ModelPath + '/InstantMessengerModel.uml'),
    parent <- root)
}
```

The "ConfigRoot" rule matches against the root element of each configuration model, which is always an instance of "InstantMessengerConfiguration". This rule triggers several other called rules, including a rule that generates a boilerplate Ant script header ("ConfigHeader"), a rule that defines the "all" target that performs a complete build ("AllTarget") and a rule that generates an "InstantMessengerModel" target ("ModelTarget") for building the main "InstantMessenger" feature (see Fig. 5.2). The "ConfigHeader" rule is defined as follows:

```
rule ConfigHeader(s : CFG!"transformations::TransformationConfig") {
  -- <project name="s.buildPath">
  to name : XML!"XML::Attribute" (
      parent <- s,
      name <- 'name',
    value <- s.buildPath),
```

```
  -- <import file="common.xml"/>
     import : XML!"XML::Element" (
      parent <- s,
      name <- 'import'),
     importfile : XML!"XML::Attribute" (
      parent <- import,
    name <- 'file',
    value <- 'common.xml'),
  -- <property name="current.path" value="/${ant.project.name}"/>
     currentpath : XML!"XML::Element" (
      parent <- s,
    name <- 'property',
    children <- thisModule.Property('current.path', '/${ant.project.name}')),
  -- <property name="saveNoModels|saveModels|saveLastModel" value="true"/>
     save : XML!"XML::Element" (
      parent <- s,
    name <- 'property',
    children <- thisModule.Property(
      if s.saveModels = #none then 'saveNoModels' else
      if s.saveModels = #all then 'saveModels' else
      'saveLastModel' endif endif,
      'true'))
  do {
    'project';
  }
}
```

This rule generates the Ant script header. An example of a generated Ant script header is given below:

```
<project name = 'uml2cs-instantmessenger-default/build' default = 'all'>
  <import file = 'common.xml'/>
  <property name = 'current.path' value = '/${ant.project.name}'/>
  <property name = 'saveLastModel' value = 'true'/>
```

Note that an "import" tag is generated, which imports the Ant code from "common.xml". The "common.xml" file contains all static Ant code that does not need to be generated by a model transformation[2]. If one compares the generated Ant code to the transformation rule that generates it, one can see that the generated code is a number of times smaller than the transformation rule. It is therefore interesting to not generate more code than strictly necessary.

The "ConfigHeader" rule uses another rule named "Property", which is defined as follows:

```
rule Property(name : String, value : String) {
  -- Sequence{name="name" value="value"}
  to propertyname : XML!"XML::Attribute" (
      name <- 'name',
    value <- name),
     propertyvalue : XML!"XML::Attribute" (
      name <- 'value',
    value <- value)
  do {
    Sequence{propertyname, propertyvalue};
  }
}
```

[2]The "common.xml" file can be found at http://ssel.vub.ac.be/viewvc/UML2CaseStudies/uml2cs-transformations/common.xml?revision=7297&view=markup

The "ModelTarget" rule generates an Ant target that builds a specific feature, such as our main "InstantMessenger" feature. It is defined as follows:

```
rule ModelTarget(name : String, inModel : String, prefix : String,
    path : String) {
  -- Sequence{name = 'name' depends = 'allBase'}
  to targetName : XML!"XML::Attribute"(
      name <- 'name',
    value <- name),
     targetDepends : XML!"XML::Attribute"(
      name <- 'depends',
    value <- 'allBase'),
  -- <am3.loadModel modelHandler = 'EMF' name = 'inModel'
  --  metamodel = 'UML2' path = 'path'/>
     loadModel : XML!"XML::Element" (
      name <- 'am3.loadModel'),
     modelHandler : XML!"XML::Attribute" (
      name <- 'modelHandler',
    value <- 'EMF',
    parent <- loadModel),
     modelName : XML!"XML::Attribute" (
      name <- 'name',
    value <- inModel,
    parent <- loadModel),
     metamodel : XML!"XML::Attribute" (
      name <- 'metamodel',
    value <- 'UML2',
    parent <- loadModel),
     modelPath : XML!"XML::Attribute" (
      name <- 'path',
    value <- path,
    parent <- loadModel),
  -- <antcall target="generate" inheritRefs="true">
     antcall : XML!"XML::Element" (
      name <- 'antcall'),
     antcalltarget : XML!"XML::Attribute" (
      name <- 'target',
    value <- 'generate',
    parent <- antcall),
     inheritRefs : XML!"XML::Attribute" (
      name <- 'inheritRefs',
    value <- 'true',
    parent <- antcall),
  -- <param name="in" value="inModel"/>
     paramin : XML!"XML::Element" (
      name <- 'param',
    children <- thisModule.Property('in', inModel),
    parent <- antcall),
  -- <param name="prefix" value="prefix"/>
     paramprefix : XML!"XML::Element" (
      name <- 'param',
    children <- thisModule.Property('prefix', prefix),
    parent <- antcall)
  do {
    Sequence{targetName, targetDepends, loadModel, antcall};
  }
}
```

This rule generates an Ant target. An example of a generated Ant target for our main "InstantMessenger" feature is given below:

```
  <target name = 'InstantMessengerModel' depends = 'allBase'>
    <am3.loadModel modelHandler = 'EMF' name = 'inModel' metamodel = 'UML2'
         path = '/uml2cs-instantmessenger-model/models/InstantMessengerModel.uml'/>
```

```
    <antcall target = 'generate' inheritRefs = 'true'>
      <param name = 'in' value = 'inModel'/>
      <param name = 'prefix' value = 'im'/>
    </antcall>
  </target>
```

The Ant target depends on "allBase", which is defined in "common.xml". It then loads the input model and calls the "generate" target with the loaded model as a parameter. The "generate" target is also defined in "common.xml" and will generate the code for the given model.

The transformation also includes a (matched) transformation rule for each feature meta-class in the meta-model. The transformation rule for the "LocalNetwork" feature, for example, looks like this:

```
rule LocalNetwork {
  from s : CFG!"instantmessenger::LocalNetwork"
  -- <target name = 'LocalNetwork' depends = 'allBase'>
  to target : XML!"XML::Element" mapsTo s (
    name <- 'target',
    children <- thisModule.ModelTarget(
      'LocalNetwork',
      'Local',
      'local',
      thisModule.ModelPath + '/InstantMessengerLocal.uml'),
    parent <- s.config)
}
```

If triggered, the rule generates the following Ant code:

```
  <target name = 'LocalNetwork' depends = 'allBase'>
    <am3.loadModel modelHandler = 'EMF' name = 'Local' metamodel = 'UML2'
        path = '/uml2cs-instantmessenger-model/models/InstantMessengerLocal.uml'/>
    <antcall target = 'generate' inheritRefs = 'true'>
      <param name = 'in' value = 'Local'/>
      <param name = 'prefix' value = 'local'/>
    </antcall>
  </target>
```

Note how the generated Ant target looks very much like the Ant target for the main "InstantMessenger" feature. The "generate" target is again invoked, but this time the input model is different. The input model actually contains the "LocalNetwork" feature.

The "AllTarget" rule finally generates the Ant target that performs the complete build. It is defined as follows:

```
rule AllTarget(c : CFG!"instantmessenger::InstantMessengerConfiguration") {
  -- Sequence{name="all" depends="..."}
  to targetName : XML!"XML::Attribute"(
      name <- 'name',
    value <- 'all'),
     targetDepends : XML!"XML::Attribute"(
      name <- 'depends',
    value <- Sequence{'InstantMessengerModel'}
        ->union(c.localNetworkDepends())
      ->union(c.jabberNetworkDepends())
      ->union(c.userInterfaceDepends())
      ->iterate(e; acc : String = '' | acc +
        if acc = '' then e
```

```
        else ', ' + e endif).debug('All')),
  --  <eclipse.refreshLocal resource="${ant.project.name}"/>
     refresh : XML!"XML::Element" (
      name <- 'eclipse.refreshLocal'),
     refreshResource : XML!"XML::Attribute" (
      name <- 'resource',
    value <- '${ant.project.name}',
    parent <- refresh)
  do {
    Sequence{targetName, targetDepends, refresh};
  }
}
```

Depending on the features that were selected through the configuration model, the "AllTarget" rule generates the following Ant code:

```
<target name = 'all' depends = 'InstantMessengerModel, LocalNetwork, JabberNetwork,
       DefaultJabberTransport, AWTUserInterface'>
  <eclipse.refreshLocal resource = '${ant.project.name}'/>
</target>
```

The generated "all" target makes sure that the targets for all selected features are executed. It does this by adding all relevant targets to its "depends" attribute. The "AllTarget" rule uses a number of helper methods to determine which targets are indeed relevant:

```
helper context CFG!"instantmessenger::InstantMessengerConfiguration"
def : localNetworkDepends() : Sequence(String) =
  if self.localNetwork.oclIsUndefined() then
    Sequence{}
  else
    Sequence{'LocalNetwork'}
  endif;

helper context CFG!"instantmessenger::InstantMessengerConfiguration"
def : jabberNetworkDepends() : Sequence(String) =
  if self.jabberNetwork.oclIsUndefined() then
    Sequence{}
  else
    if self.jabberNetwork.jabberTransport.oclIsKindOf(
      CFG!"instantmessenger::MEJabberTransport") then
      Sequence{'JabberNetwork', 'MEJabberTransport'}
    else
      Sequence{'JabberNetwork', 'DefaultJabberTransport'}
    endif
  endif;

helper context CFG!"instantmessenger::InstantMessengerConfiguration"
def : userInterfaceDepends() : Sequence(String) =
  if self.userInterface->select(u|u.oclIsKindOf(
    CFG!"instantmessenger::AWTUserInterface"))->notEmpty() then
    Sequence{'AWTUserInterface'}
  else
    Sequence{}
  endif
  ->union(
  if self.userInterface->select(u|u.oclIsKindOf(
    CFG!"instantmessenger::SwingUserInterface"))->notEmpty() then
    Sequence{'SwingUserInterface'}
  else
    Sequence{}
  endif)
```

```
->union(
if self.userInterface->select(u|u.oclIsKindOf(
  CFG!"instantmessenger::LCDUIUserInterface"))->notEmpty() then
  Sequence{'LCDUIUserInterface'}
else
  Sequence{}
endif);
```

The complete Ant script now contains a default "all" task that generates code for the entire product configuration. Note that not every step in the product building process needs to be automated. Packaging, testing and deployment, for example, is often done manually.

5.5 Summary

In this chapter, we have explained what a Software Product Line (SPL) is and how it can be organised. Three main SPL activities have been discussed, as well as their resulting artifacts: Commonality and Variability Analysis (CVA), Feature Modelling and Configuration. The example of the MDA-based instant messenger is used to illustrate these three activities. Finally, we showed how model-driven techniques can be used for product configuration.

The next chapter discusses how the configuration of MDA-based SPLs can be done.

Chapter 6

Configuration of MDA-based product lines

6.1 Introduction

In chapter 4, we have discussed how MDA can be improved by using explicit platform models that allow us to reason about platform dependencies introduced by model transformations and other software artifacts. While we have shown that our platform models can be used to support the configuration of MDA model transformations, we have not discussed how this configuration is done. In fact, MDA itself does not cover the topic of configuration, even though this is necessary to manage the variability caused by platform diversity.

Software Product Lines (SPLs), on the other hand, have a long history of dealing with commonality and variability, where platform diversity can be seen as yet another variability. SPLs typically integrate a number of software-intensive products that share a significant amount of functionality. As such, any software that is developed using the MDA approach can be considered as an SPL, since each platform-specific software product shares significant functionality with other platform-specific versions of that software product. This shared functionality typically resides in the PIM and/or CIM. We refer to any software that is developed (partly) using the MDA as *MDA-based SPLs*.

This chapter discusses how the configuration of MDA-based software product lines is organised. Section 6.2 motivates the need for an integrated configuration approach for MDA and SPLs. It first looks at current SPL technology and shows how far this technology brings us in dealing with diverse platforms. Remaining issues are discussed and the need for integration of MDA and SPL technology is explained. Section 6.3 explains how models can be used to express and enforce configuration rules. Section 6.4 discusses how configuration models can be integrated with our platform ontologies. Section 6.6 discusses related work and section 6.7 concludes this chapter.

6.2 Managing MDA configurations

Even though variability is always present in the MDA in the form of various platforms, the topic of configuration is not covered by the MDA itself. As has been discussed in the previous chapter, model transformations are currently used for one platform only, which reduces the configuration task to simply selecting a platform. As the reuse of model transformations for multiple platforms has been made possible by our platform modelling approach, more elaborate configurations of model transformations are now possible. In addition to having platform dependency constraints, model transformations can also have mutual constraints. In [MTR05], Mens et al. explain the kind of conflicts that can arise between model transformations. They also propose a solution based on critical pair analysis of graph transformations.

The problem of conflicting configuration choices is also known in SPLs as the *feature interaction* problem [RMR05]. The fact that the SPL community has come up with several solutions to the feature interaction problem and other issues related to configuration, indicates that the MDA can really benefit from the knowledge built up by the SPL community.

The elements of the MDA map well to SPLs: in chapter 5, we have used the instant messenger example to demonstrate that models can represent features of a product line. In the MDA, the features are not only introduced by models that represent specific functionality (e.g. a Jabber protocol plug-in for an instant messaging client), but also by model transformations that integrate platform-specific information into PSMs. We therefore choose to also describe the MDA model transformations as features for the purpose of configuration.

We propose to use feature models to record the analysis of scope, commonality and variability of MDA-based SPLs, as it is a proven method for communicating such information. For the configuration, we propose to use Domain-Specific Modelling Languages (DSMLs), as it is a well-known and proven configuration approach for SPLs with mature and extensible tool support. The overlap between DSML and MDA technology means that they benefit from each other's advances and it reduces the required technological scope of our approach.

DSMLs use meta-models to express the rules of the configuration language. The scope of these rules is always limited to the language vocabulary. The language vocabulary typically consists of the elements that must be configured (represented as meta-classes). This means that the configuration language rules can only express which elements can be combined (feature interaction). External factors, such as platform elements, are not part of the meta-model and are hence beyond the scope of the configuration rules.

That's why we propose to extend the meta-models of configuration languages with platform dependency constraint annotations. This allows us to link configuration language elements, which correspond to configuration

choices, to platform dependency constraints, which are stored in a separate OWL ontology. In addition, an intermediate "shadow" model is used to record ordering information about configuration elements with platform dependency constraints. This ordering information must be recorded to support the manual part of our *most-specific-first* and *least-specific-first* optimisation process discussed in the previous chapter. Using a "shadow" model instead of adding more annotations to the configuration language meta-model makes it easier to translate our approach to other configuration solutions than DSMLs (e.g. feature models, grammar-based DSLs or automated logic-based approaches [BBRC06]). Only the minimal extension of mapping configuration language meta-classes to platform dependency constraints needs to be translated to the new solution domain.

6.3 Using models for configuration management

Models are playing an increasingly important role in SPLs. Feature models are used to record the scope, commonality and variability analysis result and Product Models are used to automatically generate products from the SPL's core assets. Especially the Product Models are becoming richer as more of the product derivation process becomes automatic [Kru06]. We will first discuss how to model MDA-based software artefacts as features. After that, we discuss how to derive a configuration DSML that can be used to model products.

6.3.1 Feature modelling for the MDA

In chapter 5, we have discussed an example SPL of an instant messenger. Fig. 5.2 shows the feature model for this example. We now extend the example feature model to include the PIM-to-PSM model transformations as features. The extended feature model is shown in Fig. 6.1.

The extended feature model adds an extra "TransformationConfig" feature, which represents the configuration of PIM-to-PSM model transformations that is required for a specific product in the instant messenger product line. "TransformationConfig" is actually a placeholder feature for all its subfeatures, since it contains too many subfeatures to display within the same diagram. Therefore, the subfeatures are shown in Fig. 6.2. This has the additional advantage of making the "TransformationConfig" feature tree reusable for other SPLs than the instant messenger SPL.

The subfeatures of "TransformationConfig" represent the various model transformations that are required, as well as two sets of alternative model transformations ("Observer" and "Applet"). Almost all of these model transformations require a "Mapping" to be selected. The "Mapping" feature is

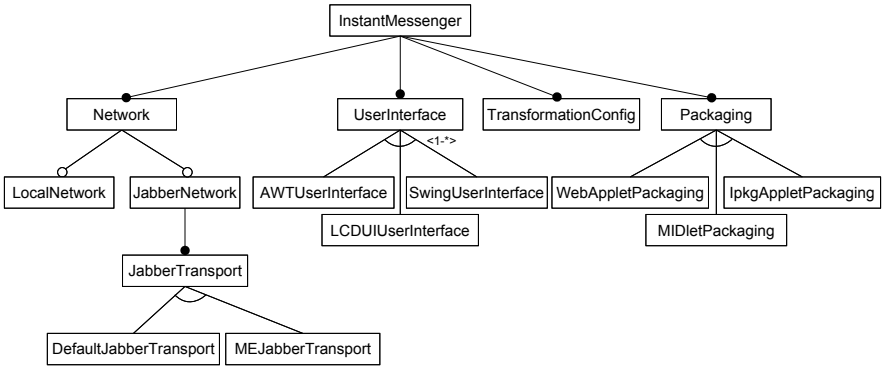

Figure 6.1: The extended feature model of the instant messenger product line.

modelled as a *shared* subfeature of these model transformations. The shared "Mapping" subfeature is mandatory in all cases.

There is an extra constraint on the configuration of the "Mapping" feature in that each model transformation must use the same mapping to achieve a consistent transformation result. If one transformation uses Java language mappings, then all transformations must use Java language mappings. If one transformation uses "Java1DataTypes", all transformations must use those data types to guarantee type consistency. Unfortunately, there is no easy way to express this constraint in the feature diagram itself. It must be recorded separately as an annotation to the feature diagram, which can take the form of a logic statement or natural language. Since our feature model is only used to derive the configuration language meta-model, this shortcoming is of minor importance for us.

It is now clear that a separate feature model for the model transformations is desired here, because the model transformations are applicable for any class of software that runs as an applet (or MIDlet). The feature model for the model transformations may therefore be reused for other applet-style software than instant messengers.

6.3.2 Configuration DSMLs

As has been explained in chapter 5, the meta-model for the DSML can be derived from the feature model. The features from the feature model that are not relevant for configuration purposes, such as the mandatory features, are omitted from the DSML meta-model. We also split the model transformation configuration and the specific instant messenger configuration in two separate meta-models.

This follows the structure of the feature model and reflects the difference of scope between model transformation configuration and instant messenger

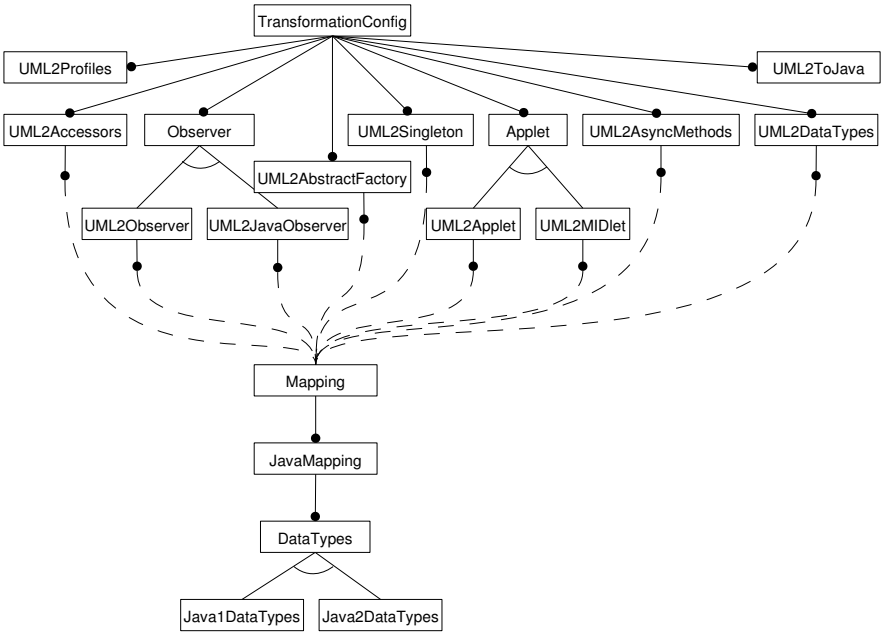

Figure 6.2: The feature model of the TransformationConfig feature.

configuration. Fig. 6.3 shows the meta-model for the configuration of the instant messenger product line example.

The separate meta-models for model transformation configuration and instant messenger configuration are shown as separate packages: "transformations" and "instantmessenger". "InstantMessengerConfiguration" is modelled as a special case of a "TransformationConfig". None of the mandatory features ("UML2Profiles", "UML2Accessors", ...) are modelled, since they are always configured to be included. In the case of alternative features, such as "UML2-Observer" / "UML2JavaObserver" and "DefaultJabberTransport" / "MEJabberTransport", the different alternatives can be modelled through subclassing of a general meta-class. "DefaultJabberTransport" and "MEJabberTransport" are subclasses of the abstract "JabberTransport" meta-class. The association between "JabberNetwork" and "JabberTransport" enforces that exactly one of the "JabberTransport" subclasses must be chosen. In the case of "UML2-Observer" and "UML2JavaObserver", a shortcut is taken: "UML2Observer" is used to represent the general case and "UML2JavaObserver" is a subclass of "UML2Observer". This approach yields the same effect, since exactly one instance of "UML2Observer" or "UML2JavaObserver" must be chosen to satisfy the association between "TransformationConfig" and "UML2Observer".

In the "TransformationConfig" feature model (Fig. 6.2), many features include the mandatory "Mapping" subfeature. We have indicated that there is an extra constraint on the configuration of the "Mapping" feature in that

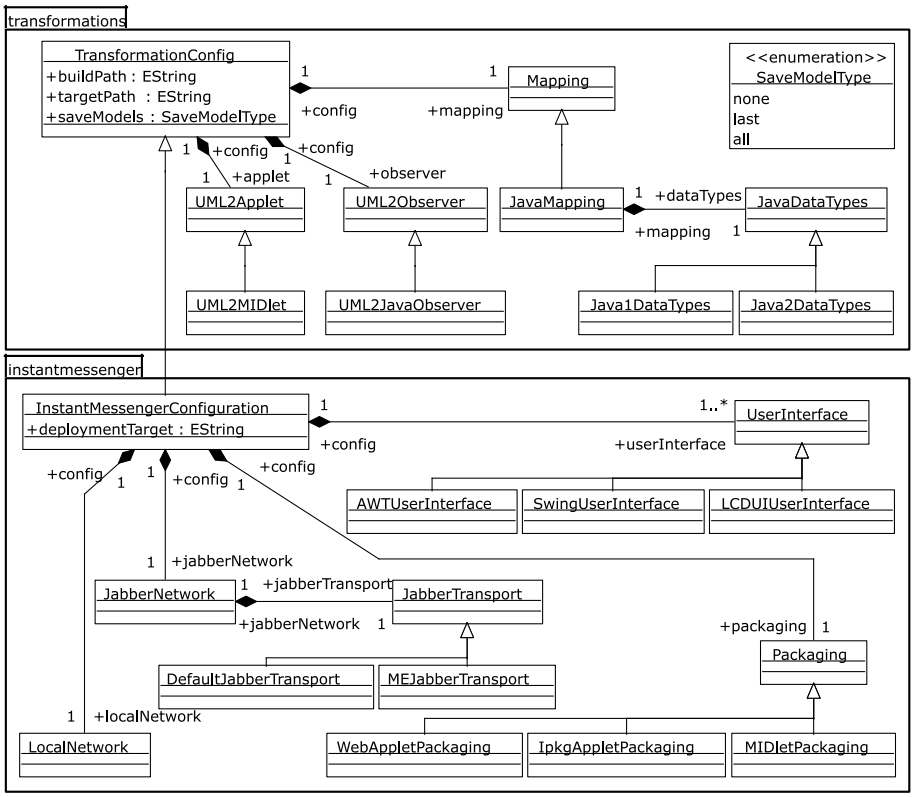

Figure 6.3: The meta-model of the instant messenger configuration language.

each model transformation must use the same mapping to achieve a consistent transformation result. If one transformation uses Java language mappings, then all transformations must use Java language mappings. If one transformation uses "Java1DataTypes", all transformations must use those data types to guarantee type consistency. While this was difficult to incorporate in the feature model, it is very much possible to include this constraint in the configuration language meta-model. The configuration choices for the "Mapping" feature are really made only once. That is why the meta-model of the transformation configuration language enforces this choice to be made exactly once through the association between "TransformationConfig" and "Mapping".

Note that our style of deriving a meta-model from a feature model results in a language that requires minimal user input to achieve a complete configuration. Contrast this to the way in which most of the automated analysis approaches work (see chapter 5, subsection 5.3.1): they focus on expressiveness of feature interaction constraints. Complex feature interaction constraints should be limited to a minimum in a configuration language meta-model, however. If not, the act of configuration becomes a guessing game where the software engineer has to find out which complex combinations are allowed by the meta-model. Instead, the meta-model should be tuned towards deriving

a configuration using as little information as possible, much like the way an expert system works. This is why the meta-classes correspond to configuration choices rather than features. Any remaining complex interactions between configuration choices can be expressed using OCL, as is explained in chapter 5.

6.4 Platform-aware configuration

In order to leverage the platform model in the configuration process, a link has to be made from the configuration language to the platform model. When looking at the configuration language meta-model shown in Fig. 6.3, it becomes apparent that each concrete meta-class can impose certain platform dependency constraints. Whenever a particular feature is included at least once in a configuration model, the platform dependency constraints of that type of feature – or meta-class – apply. Hence, a link to the platform model is made for each meta-class. It then becomes possible to trace back the platform dependency constraints for each configuration model through the meta-model of the configuration language.

In the Eclipse Modeling Framework (EMF), it is possible to create annotations for each model element. Annotations are grouped by name and can contain multiple key-value pairs. In our configuration meta-models, a "PlatformKit" annotation[1] has been added to each meta-class that introduces a platform dependency constraint. The platform dependency constraint itself is added to the annotation as a "PlatformConstraint" value. In addition, the meta-model itself contains a "PlatformKit" annotation, which contains an "Ontology" value that points to the platform ontology model. Table 6.1 shows the "PlatformConstraint" annotation values for each meta-class that has a platform dependency constraint. The values represent XML-style references to OWL concepts in the ontology that contains the platform dependency constraints.

The platform dependency constraint information is also recorded in an intermediate "shadow" model, called a "PlatformKit" model. The PlatformKit model keeps track of the ordering information that is used to support the manual part of our *most-specific-first* and *least-specific-first* optimisation process. PlatformKit models conform to the Ecore meta-model shown in Fig. 6.4.

The PlatformKit meta-model contains the following elements:

ConstraintSpace: Each PlatformKit model has a root "ConstraintSpace" element that can contain an ordered set of "ConstraintSet" elements. The root "ConstraintSpace" element has an "ontology" attribute that contains the relative location (URI) of the platform ontologies containing

[1]The name "PlatformKit" comes from the name of our tool; EMF annotations are usually named after the tool that uses them.

Meta-class	PlatformConstraint value
JavaMapping	#JavaMappingPlatform
Java1DataTypes	#Java1Platform
Java2DataTypes	#Java2Platform
UML2Observer	#Java1Platform
UML2JavaObserver	#JavaObserverPlatform
UML2Applet	#AppletPlatform
UML2MIDlet	#MIDletPlatform

Table 6.1: Configuration language meta-model annotations

the platform dependency constraints. The "ConstraintSpace" meta-class also defines a number of operations that are implemented by the PlatformKit tool. The "getIntersectionSet" operation returns a new "ConstraintSet" that contains references to the OWL intersection classes of each "ConstraintSet". The "getMostSpecific" and "getLeastSpecific" operations return all the "ConstraintSet" elements in *most-specific-first* and *least-specific-first* order, respectively. These operations take a boolean argument: when "true" is passed, only valid "ConstraintSet" elements are returned. The "getValid" and "getInvalid" operations return the valid and invalid "ConstraintSet" elements, respectively. A "ConstraintSet" is considered valid if all its platform dependency constraints are satisfied.

ConstraintSet: Each "ConstraintSet" in turn can consist of a number of "Constraint" elements. It is identified by the "name" attribute, which corresponds to a configuration language meta-class name if the PlatformKit model is used as a "shadow" model[2]. Just like "ConstraintSpace", this meta-class also defines a number of operations. The "isValid" operation returns whether all "Constraint" elements are valid (i.e. satisfied). The "getMostSpecific" and "getLeastSpecific" operations return the *most-specific* and *least-specific* "Constraint" element, respectively. The "getIntersection" operation returns the "Constraint" that corresponds to the OWL intersection class of all contained "Constraint" elements.

Constraint: Each "Constraint" corresponds to exactly one OWL class that represents a platform dependency constraint. The "ontClassURI" attribute contains the reference to this OWL class. The OWL class must be contained in one of the ontology files referenced by the root "ConstraintSpace" element. The "isValid" operation returns whether the cor-

[2]PlatformKit models can also be used for deployment, as will be explained later.

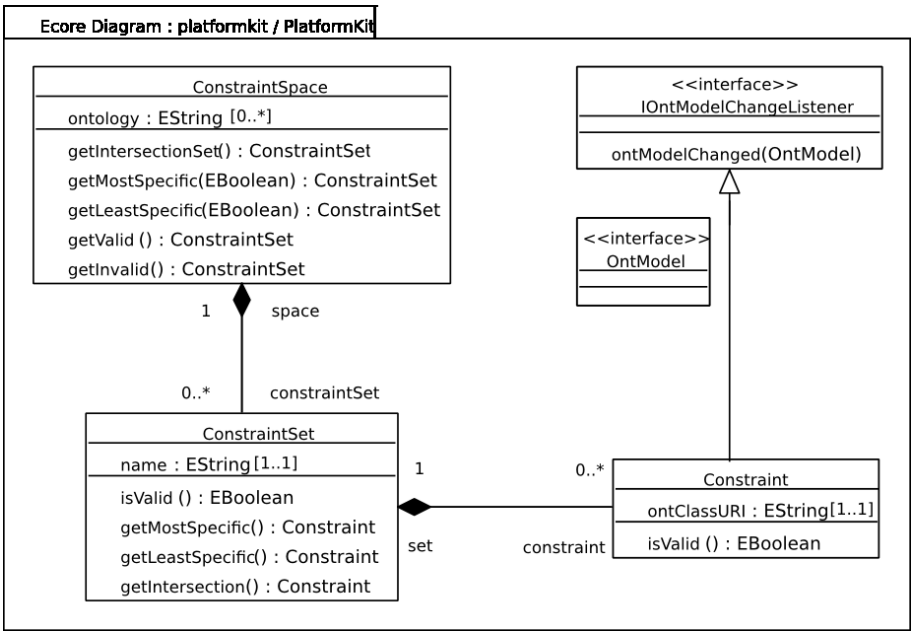

Figure 6.4: The meta-model for PlatformKit models.

responding platform dependency constraint is valid (i.e. satisfied).

We will now discuss two usage scenarios for platform dependency constraints in configuration models. First, we explain how a PlatformKit model is used to profile a configuration language meta-model against a platform instance specification. This guarantees that only those configuration elements that are valid for the given platform specification are used during the configuration activity. Second, we explain how a PlatformKit model is used to use the platform dependency constraints as a driving factor during deployment. By using a PlatformKit model that contains all available configurations, the *most-specific* or *least-specific* configuration that is still valid can be automatically selected.

6.4.1 Profiling against platform instances

One usage scenario for PlatformKit models and an annotated configuration language meta-model is to profile the meta-model against a platform instance description. In this scenario, the software engineer is able to model product configurations that are guaranteed to work on the platform that is profiled against. This scenario is described by the process flowchart depicted in Fig. 6.5.

The different automated actions are depicted as boxes, where nested boxes represent sub-actions. First, the platform dependency constraints are extracted from the configuration language meta-model and grouped per meta-class in a PlatformKit model. When using a PlatformKit model for meta-

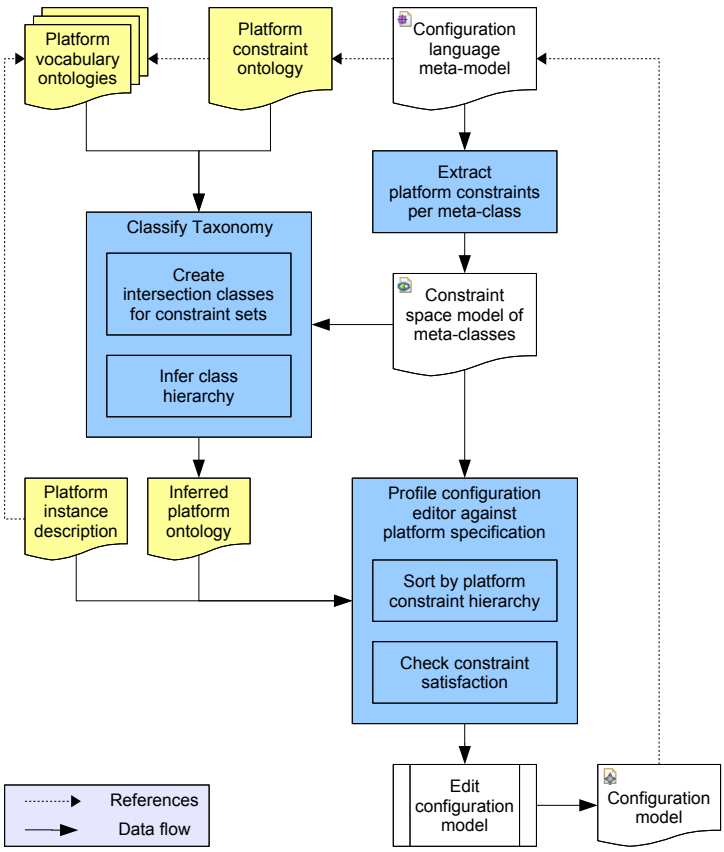

Figure 6.5: Flowchart of the platform profiling scenario

model profiling, each "ConstraintSet" in the PlatformKit model corresponds to a meta-class from the configuration language meta-model. Fig. 6.6 shows an example of a "shadow" PlatformKit model for the instant messenger configuration language. The notation used is the default notation for generated EMF editors. Each model element label starts with the name of its meta-class. The "ontology" attribute of the root "ConstraintSpace" element is not shown. Each "ConstraintSet" element displays its name in the label, which corresponds to a meta-class name from the meta-model shown in Fig. 6.3.

Together with the platform vocabulary ontologies and the platform dependency constraint ontology, the PlatformKit model provides enough information to infer the OWL class hierarchy. This action requires that the intersection classes for each "ConstraintSet" element are generated first. The resulting inferred ontology contains all relevant parts of the platform vocabulary ontologies as well as all platform dependency constraints and "ConstraintSet" intersection classes.

The PlatformKit model and inferred platform ontology are then matched – or *profiled* – against a platform instance description in OWL. The validity of

Figure 6.6: The PlatformKit model for the instant messenger configuration language.

each "ConstraintSet" within the platform specification that is profiled against can be checked. All valid and invalid "ConstraintSet" elements can be traced back to the corresponding meta-class via their names. As a result, the configuration language editor can be instructed to only allow valid meta-classes to be instantiated. Existing configuration model elements for which the meta-class is invalid can be marked. The software engineer can use this information to correct the configuration model.

Besides the ability to check which "ConstraintSet" elements – and their corresponding meta-classes – are valid, we can also sort the list of "ConstraintSet"

elements. This is done by taking the intersection class for each "ConstraintSet" and classify the hierarchy of these intersection classes. This hierarchy constitutes a partial ordering in which the leaves are considered as *most-specific* and the roots as *least-specific*. The list can be sorted *most-specific-first* or *least-specific-first* according to the process described in subsection 4.6.1 of chapter 4.

6.4.2 Platform-driven deployment

Another usage scenario for PlatformKit models and an annotated configuration language meta-model is to use the platform dependency constraint information to drive the deployment of a selected configuration. In this scenario, all relevant product configurations are checked against a platform specification and the *most-specific* or *least-specific* product configuration can be deployed. This scenario is described by the process flowchart depicted in Fig. 6.7.

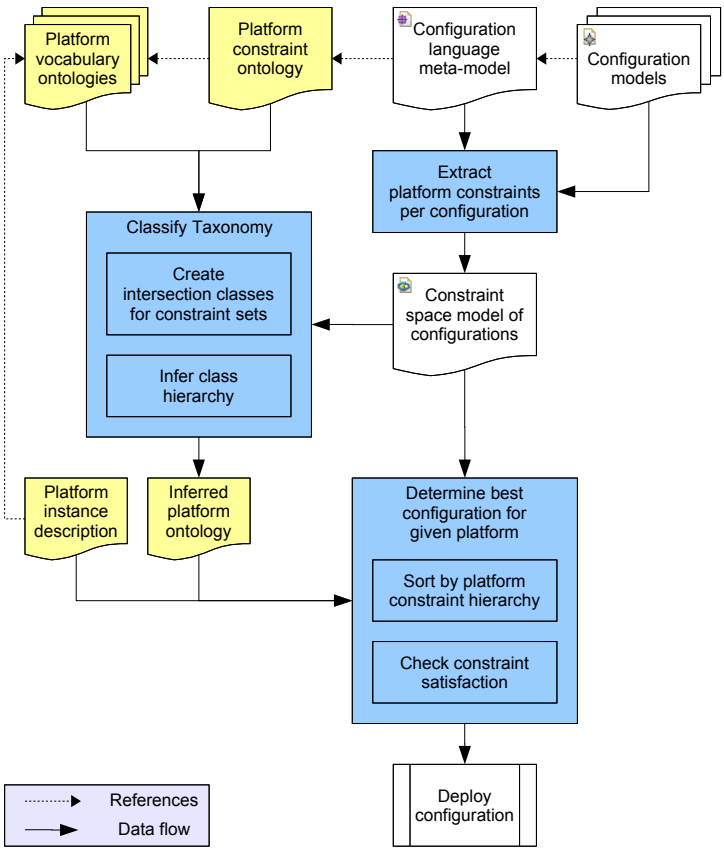

Figure 6.7: Flowchart of the deployment scenario

This scenario looks very similar to the previous one, which demonstrates the use of a "shadow" PlatformKit model. Instead of extracting all platform

dependency constraints in the configuration language meta-model, the meta-classes used in each configuration model are now retrieved and the platform dependency constraints that apply to those meta-classes are extracted. The platform dependency constraints are then grouped in the PlatformKit model as one "ConstraintSet" element per configuration model. Fig. 6.8 shows an example of a PlatformKit model for the deployment of several instant messenger configurations. Each "ConstraintSet" element displays its name in the label, which corresponds to the value of the attribute marked as "ID" from the root element in the configuration model. For the instant messenger configuration language, the root element is always an "InstantMessengerConfiguration" and the "ID" attribute is the "deploymentTarget" attribute (see in Fig. 6.3).

Inferring the OWL class hierarchy works exactly the same as in the previous scenario, except that "ConstraintSet" elements now correspond to configuration models instead of configuration language meta-classes. The validity of each "ConstraintSet" within the targeted deployment platform can be checked. All valid and invalid "ConstraintSet" elements can be traced back to the corresponding configurations via their names. The inferred OWL class hierarchy can again be used to create a sorted list of configurations that can be deployed. For the purpose of automatically selecting a product configuration for deployment, the first valid "ConstraintSet" from the list is selected and the corresponding configuration is deployed.

6.5 Limitations

Our configuration approach has certain known limitations. This section discusses these limitations and suggest how these limitations can be mitigated.

6.5.1 Model transformations are not features

Modelling transformations as features is a shortcut that works for configuration purposes, but is not valid in general. For example, a model transformation may be applied without triggering on some of the model elements or any model element. As a result, only part – or none – of the feature has been introduced. Also, less platform dependencies or no platform dependencies at all have been introduced. Selecting a model transformation does not necessarily introduce any new features in the product. Therefore, considering a model transformation equivalent to a feature in general is misleading.

To mitigate this problem, PIM-to-PSM model refinement transformations need to be considered in the scope of the PIM to which they are applied. Irrelevant transformations should not be used, as they claim extra platform dependencies that are not present in the generated PSM. As it is too much effort to develop a configuration framework for model transformations for each

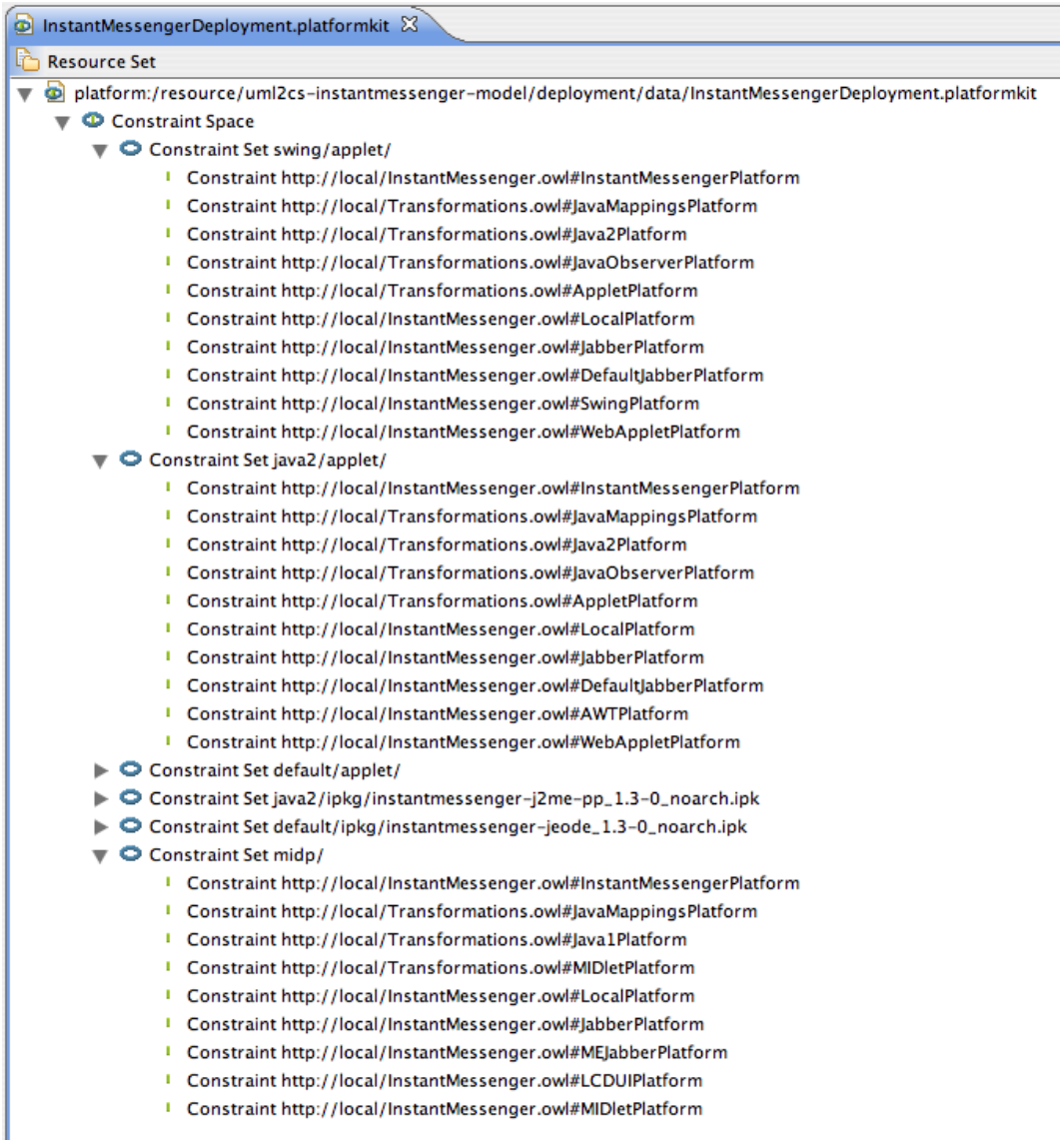

Figure 6.8: The PlatformKit model for the instant messenger product configurations.

PIM that they are applied to, it is best to have a number of model transformation frameworks – including configuration support – that can be applied within a certain scope. Our model transformations, for example, are tailored to the scope of Java applets and MIDlets. A number of those transformations could be reused and augmented with other transformations to create a transformation framework for Java web applications.

6.5.2 Scalability

The fact that we have chosen DSMLs for configuration has consequences for scalability. As the amount of features – including model transformations – rises, not only the DSML meta-model needs to be extended, but also the model transformations that generate the configuration implementation must be updated. Since those transformations can quickly grow more complex, this limits the amount of features that can still be reasonably maintained.

This problem can be mitigated by focusing the model transformation that generates the configuration implementation to only do translation of configuration choices instead of generating the actual implementation. Let's assume our configuration is implemented by an Ant build script that executes the different PIM-to-PSM refinement transformations on the selected PIMs (see chapter 5). Since most of the build workflow is fixed, this can be encoded in a static Ant script that uses parameters to decide between alternative workflows. Those parameters can then be provided in a small generated Ant script that invokes the static Ant script. This considerably reduces the size and complexity of the build script generator model transformation.

Another problem that arises when the number of features increases is that the rules for combining the different features may grow out of hand and become inconsistent. This problem applies not only to our chosen DSML approach, but also feature models themselves. There are currently a number of formal configuration approaches that can detect inconsistencies in the configuration rules that result in so-called *dead features*: features that can never be included in any configuration without breaking configuration rules. These approaches are based on propositional logic and constraint programming and are described by Benavides et al. in [BRCTS06].

6.6 Related work

We chose to use domain-specific modelling as our configuration management approach. In [CHE05b], it is demonstrated that *feature models* can also be used for configuration purposes. In this context, a *configuration* consists of the features that were selected according to the variability constraints defined by the feature model. The relationship between feature modelling and domain specific languages is explored in [DK02]. An important conclusion is that feature models can be translated into a DSL grammar (or meta-model). This is illustrated by their Feature Description Language (FDL), which follows the same structure as a BNF grammar. The meta-models we use for describing our features can hence be considered equivalent to feature models. The fact that both DSMLs and the MDA itself can share the same model-driven technology also greatly simplifies matters.

In [AC06], Antkiewicz and Czarnecki introduce Framework-Specific Modelling Languages (FSMLs). FSMLs are used to generate framework extension code from a domain-specific model. It is comparable to a project creation wizard in the Eclipse framework, but more powerful with respect to changing your parameters and re-generating the framework extension code. FSMLs support round-trip engineering and propagate changes to the model down to the code as well as propagate changes to the code back to the model (where applicable). FSMLs are based on DSMLs and are related to the way we use DSMLs in the sense that we both use models to "configure" a code generator. This means that for both approaches a lot of domain knowledge is part of the code generator. For FSMLs, the code generator knows how to extend the framework. For our configuration languages, the code generator knows how and in which order to apply PIM-to-PSM refinement transformations.

There also exist specific approaches for the configuration of model transformations in the MDA. In [Old05], Oldevik describes a framework for modelling compositions of model transformations, which is based on UML. The approach focuses on workflow descriptions, which essentially describe the execution order of transformations. It does not describe alternative transformation workflows, but it seems possible to extend the framework for this. Oldevik's composition model may add to the traditional "configuration-model-to-build-script" generative approach described in chapter 5 by adding a higher-level workflow description of how the model transformations should be executed. Instead of generating low-level build script code, we can generate a more concise workflow description model. The composition framework then executes that workflow.

In the field of Software Product Lines, various research exists on the automated analysis of feature models [BBRC06]. This research focuses on the analysis of *feature interaction* constraints [RMR05] however, whereas our focus lies on platform dependency constraints that refer to the context of the features rather than the features themselves. This means that our approach forms an addition to these automated analysis methods for feature models. Especially the approach described by Wang et al. in [WLS+07] fits well with our approach, since it also use OWL DL to describe feature interaction constraints and we can reuse the knowledge of OWL that is necessary to use our approach. The approach described by Benavides et al. in [BTRC05], which is based on constraint programming, promises more powerful analysis results, however [BRCTS06]. We use OCL to express complex constraints on the metamodel of our configuration language, which offers sufficient expressiveness for our purposes. The expressiveness of OCL hasn't been pinpointed precisely, however. In [MC99], Mandel and Cengarle claim that OCL cannot be considered equivalent to either relational calculus or Turing machines. It is also not (yet) possible to detect inconsistencies between OCL constraints, which is a powerful analysis tool for feature modelling.

In [WSWN07], White et al. describe a method for automatically selec-

tion SPL variants for mobile devices. Their configuration approach is identical to ours, where a DSML serves as a configuration language. They acknowledge that, in addition to feature interaction constraints, target device constraints must be taken into account as well. These target device constraints have the same purpose as our platform dependency constraints. White's target device constraints focus mostly on hardware resources, however, where we have focused on software resources. An example target device constraint is "`JVMVersion > 1.2`" or "`WifiCapable = true`". In comparison, our platform dependency constraints are much more detailed than "`JVMVersion > 1.2`". The method White et al. use for SPL variant selection is based on a Prolog constraint solver and is actually based on Benavides' work [BTRC05]. Optimisation is based on a developer-supplied cost function that is applied to all remaining valid variants. Compare this to our approach, where optimisation is based on constraint classification, is done offline and before determining constraint satisfaction. Performance of both White's approach and our approach is comparable with our current use of DLs and associated reasoners ($\mathcal{SHF}(\mathbf{D})$ on a $\mathcal{SHOIN}(\mathbf{D})$ reasoner – see chapter 4).

6.7 Summary

This chapter has indicated that the topic of configuration is hardly covered by the MDA, since with the traditional monolithic PIM-to-PSM transformations configuration is limited to selecting a target platform. As one starts using multiple successive PIM-to-PSM refinement transformations, with alternative transformations depending on the targeted platform, managing the transformation configuration becomes important.

The field of Software Product Lines, in contrast, have long dealt with software variability, whether caused by platform diversity or other factors. Feature models describe the common and variable parts of an SPL. We have shown that PIM-to-PSM refinement transformations can be considered as features too in the context of a feature model. Whereas feature models are very suitable for analysis purposes, we have chosen to use Domain-Specific Modelling Languages as a configuration tool. DSMLs form a proven and flexible method to describe SPL configurations. MDA technology for meta-modelling and model transformation can be reused for the development of a DSML. The fact that a configuration language can focus on configuration only, without having to describe an analysis of the available features, makes that the language can remain simpler.

We proposed to extend the DSML configuration approach by annotating the meta-model of a configuration language with platform dependency constraints. Since each meta-class in the meta-model corresponds to a configuration choice, we add an annotation to each meta-class representing a choice that

introduces platform dependencies. These annotations refer to the platform ontology that describes the platform dependency constraints in OWL DL. The meta-model annotations allow us to extract a platform dependency constraint analysis model (PlatformKit Model) from either the meta-model directly or any configuration model (which conforms to the configuration language meta-model). Using a separate analysis language/model allows us to apply our approach to other configuration methods than DSMLs with minimal work.

We have illustrated two usage scenarios for PlatformKit Models. In the first scenario, PlatformKit Models are used to profile the configuration language meta-model against a platform instance specification. By marking the meta-classes as valid or invalid for the given platform, configuration models can be analysed on their usage of valid and invalid meta-classes. The second scenario deals with the deployment of a single configuration that is automatically selected from a group of supported configurations. In this scenario, configurations are checked against a platform instance specification as a whole. Configurations can be sorted *most-specific-first* or *least-specific-first*, then checked for validity for the given platform.

Two known limitations of our chosen configuration approach have been discussed: the mismatch between model transformations and features and scalability with regard to the number and complexity of features supported. Finally, this chapter has shown where our work is situated with respect to related work in the area of configuration for the MDA and SPLs.

The following chapter introduces our PlatformKit tool. PlatformKit provides support for reasoning about platform models and configuration based on platform dependency constraints.

Chapter 7

Tool support

7.1 Introduction

A proof-of-concept tool, called PlatformKit[1], has been developed that implements platform dependency constraint checking as well as platform-based optimisation. PlatformKit provides an Eclipse[2] plug-in that provides developers with platform dependency constraint management support. It uses EMF for all meta-modelling and ATL for its transformations. It uses Jena [3] for OWL-DL manipulation. The tool uses a DL reasoner, such as Pellet [SPG+07], Fact++ [TH06] or Racer [MH03], for the classification of the constraint taxonomy as well as constraint validation. Modelling platform dependency constraint ontologies is done using the Protégé tool[4], using our platform vocabulary as a basis.

PlatformKit also provides a web interface (Java servlet) to support platform-based deployment of the various configurations. The servlet receives a platform instance specification from a client, against which the available configurations are validated. It can either return the *most-specific* (default) or *least-specific* option that is still valid, based on the client's preferences. It then redirects the client to the URL that contains the corresponding configuration.

Apart from PlatformKit, work has been done on other tool support. The Jar2UML[5] tool was developed to reverse engineer detailed UML models of Java API and Java dependencies. For our instant messenger case study, we've made several changes to the ATLAS Transformation Language tool support. These changes include support for modularised meta-models, such as Eclipse UML2, and superimposition.

[1] http://ssel.vub.ac.be/ssel/research:mdd:platformkit
[2] http://www.eclipse.org/
[3] http://jena.sourceforge.net/
[4] http://protege.stanford.edu/
[5] http://ssel.vub.ac.be/ssel/research:mdd:jar2uml

The remainder of this chapter is structured as follows: section 7.2 describes the architecture of the PlatformKit components. Section 7.3 explains all the tasks that can be performed by the tool. Section 7.5 discusses the limitations of PlatformKit and provides directions for future work. Finally, section 7.6 concludes this chapter.

7.2 Architecture

PlatformKit consists of two components, each of which make use of several third-party components. Fig. 7.1 gives an overview of those components and how they depend on eachother.

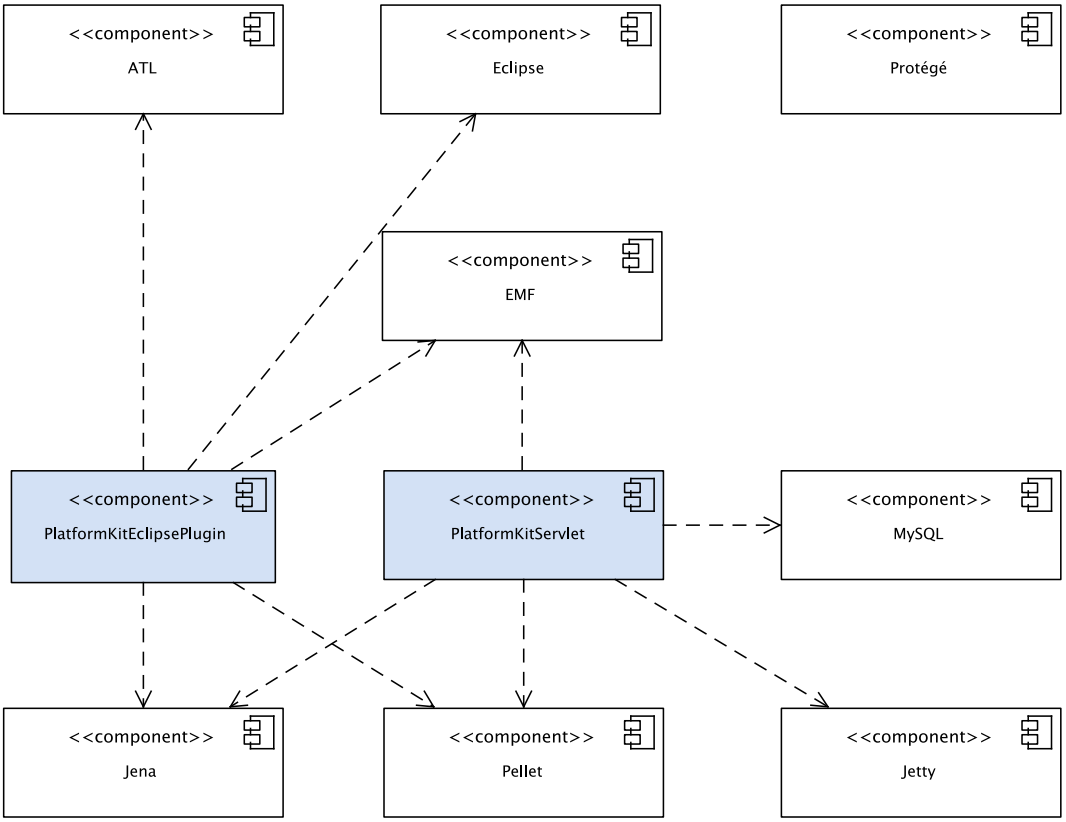

Figure 7.1: PlatformKit architectural overview

The PlatformKit Eclipse plug-in adds functionality to the Eclipse Integrated Development Environment for the management of platform dependency constraints. It uses the Jena framework for the manipulation of platform ontologies. The Pellet DL reasoner is used as the built-in component for all reasoning tasks. PlatformKit also allows the use of an external DIG-compliant[6]

[6] http://dl.kr.org/dig/interface.html

reasoner. EMF is used as a basis for the Platformkit Model language (see chapter 6) that tells PlatformKit what it must reason about. EMF is also used as the basis for configuration editors: PlatformKit can instrument any EMF-based model editor to validate against platform dependency constraints. ATL is used to run queries against UML models of the various Java APIs. This specific functionality is used to measure the platform dependencies of third party Java components or libraries.

The PlatformKit servlet implements web-based deployment based on platform dependency constraints. It also uses EMF to read the Platformkit Models and it uses Jena and Pellet to reason about platform dependency constraints. In addition, the PlatformKit servlet uses MySQL[7] to store a table of known standard platforms. The PlatformKit servlet itself is deployed on a Jetty application server[8].

The two PlatformKit components implement the scenarios that are explained in section 6.4 of chapter 6. Both components use a generated Java implementation of the PlatformKit model (see Fig. 6.4) that has been refined to implement the functionality of the specified operations (getIntersectionSet, getMostSpecific, etc.). The implementation of PlatformKit can be found at http://ssel.vub.ac.be/viewvc/PlatformKit/.

7.2.1 Jar2UML

PlatformKit also makes use of the UML models that are generated by our Jar2UML reverse engineering tool. Existing reverse engineering tools, such as Netbeans[9], MaintainJ[10] and Altova[11], were not suitable to retrieve the necessary information from Java binaries, such as a Java class library. They were typically intended for reuse or understanding of existing Java source code or byte code. The focus of Jar2UML is on providing a tailored UML model for PlatformKit that provides exactly those details that are necessary to determine (binary) compatibility between APIs.

The Java platform ontologies are generated from UML models (see chapter 4). In addition, UML models of jar dependencies can be compared against built-in UML models of standard Java APIs (see also subsection 7.3.2). Fig. 7.2 gives an overview of jar2uml and the components that it uses.

Jar2UML uses the Apache Bytecode Engineering Library (BCEL)[12] to read Java class files inside a jar archive. It then creates a UML model from the bytecode, including any dependencies that can be inferred from the bytecode

[7] http://www.mysql.org/
[8] http://www.mortbay.org/
[9] http://www.netbeans.org/kb/55/uml-re.html
[10] http://www.maintainj.com/
[11] http://www.altova.com/features_reverse_engineer.html
[12] http://www.apache.org/bcel/

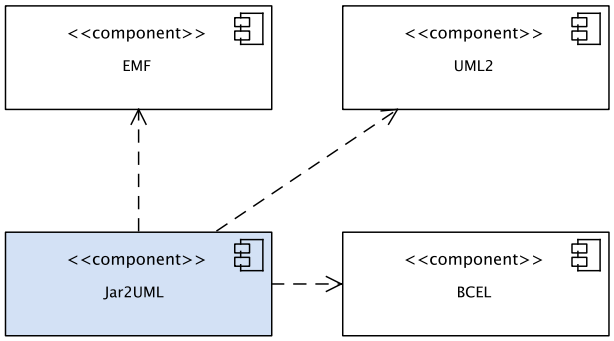

Figure 7.2: Jar2UML architectural overview

(referenced types, invoked methods, etc.). Jar2UML is part of the Eclipse MoDisco reverse engineering tool box[13]. The implementation of Jar2UML can be found at http://ssel.vub.ac.be/viewvc/JarToUML/.

7.3 Tasks

PlatformKit can be used to support several tasks. These tasks are also described in the PlatformKit manual[14]. This section describes the tasks from a usage point of view; installation instructions, technical requirements, etc. of both the PlatformKit Eclipse plug-in and the PlatformKit Servlet can be found in the current version of the online manual. Where applicable, the tasks are illustrated using the instant messenger case study[15].

7.3.1 Setting up a Model-Driven Software Product Line

We provide a number of guidelines on the way that your MDA-based software product line is set up. These guidelines ensure that you can make full use of the functionality of the PlatformKit Eclipse plug-in. We'll assume that you use ATL for your model transformations and Java as an implementation language for your software products.

The core assets for the Instant Messenger product line are stored in an Eclipse ATL project. These core assets include at least the PIM of the common architecture for all software products as well as the PIMs of each optional feature. Any third-party components that must be built from source can be stored in a separate Eclipse Java project.

Each software product is stored in a separate Eclipse Java project. That way, project-specific settings for compilation can be used to cater for each

[13] http://www.eclipse.org/gmt/modisco/toolBox/
[14] http://ssel.vub.ac.be/ssel/research:mdd:platformkit
[15] http://ssel.vub.ac.be/ssel/research:mdd:casestudies

targeted platform (e.g. J2SE 1.5, J2ME MIDP 1.0, etc.). We can now select and apply model transformations to the common PIM and the selected optional PIMs using a build.xml Ant script. The resulting generated code can be stored in the Eclipse Java project for the relevant product configuration.

Later on we will explain how to create a configuration language for the software product line, which allows you to automatically generate a product-specific build.xml script with a model transformation. This automatic product generator is not necessary for the purpose of PlatformKit, however.

Now that the source code for the instant messenger product is available, any additional resources (e.g. icons and sound clips) can be added and packaging scripts can be written. Any reusable packaging scripts can be refactored into the main product line Eclipse project. Templates can be added to the main product line Eclipse project for product-specific building and packaging scripts.

7.3.2 Extracting Platform Dependencies of Third-party Components

Finding out the platform dependencies for your own software models and/or source code is relatively straightforward, compared to finding out the platform dependencies of third-party binaries. PlatformKit can help you to extract the platform dependencies for third-party Java jar files. It uses the UML models generated by the Jar2UML tool[16] for this.

First, we need a UML model that includes all the API dependencies of the third-party component's jar file(s):

- Select "File → Import".
- Select "Jar2UML Import → Import Jar File Dependencies to UML Models" and click "Next" (see Fig. 7.3).
- Select the component's jar file(s) by clicking "Browse". Fig. 7.4 shows how to import the MicroJabberWookie.jar file, which can be found in "uml2cs-microjabberwookie/jar/" if you installed the instant messenger case study.
- Select a target folder for your UML model.
- Click "Finish".

PlatformKit can now compare the extracted UML model against built-in UML models of various Java APIs:

[16] http://ssel.vub.ac.be/ssel/research:mdd:jar2uml

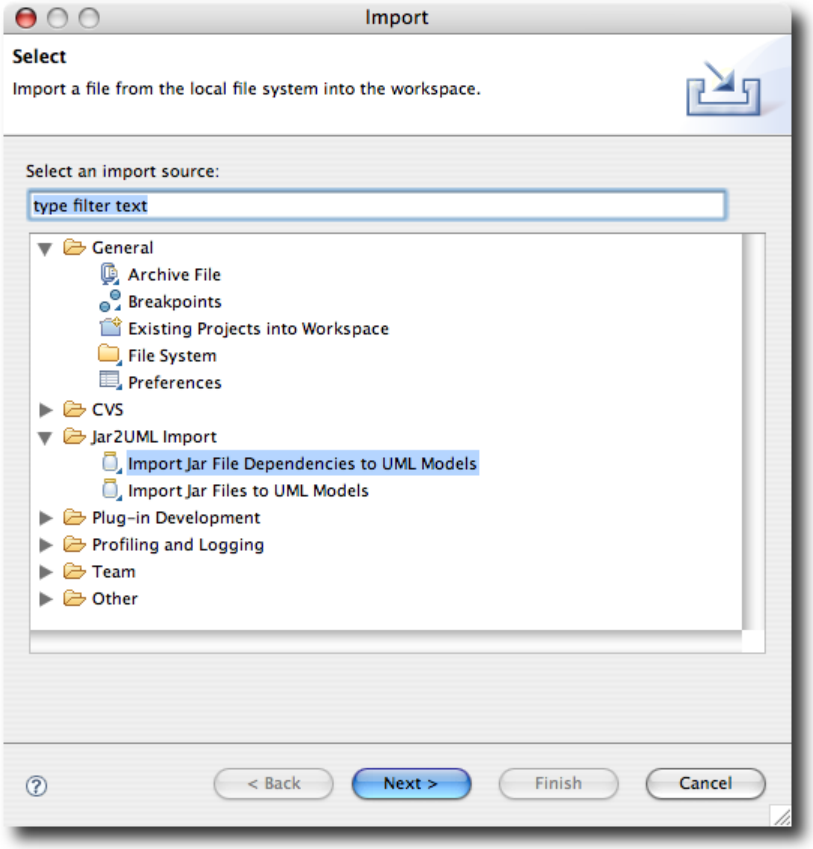

Figure 7.3: Jar2UML Dependency Import

- Right-click the ".uml" file and select "PlatformKit → Determine compatibility with X", where X is the Java API you want to compare against (see Fig. 7.5).

- An ATL query is run against the models and checks whether the selected Java API is compatible with the API dependencies we've extracted.

- The result compatible or not is displayed in a dialogue window, while the detailed comparison results can be viewed in the ATL console (see Fig. 7.6).

By finding out which Java APIs are compatible, you can derive which versions of particular Java API packages to use when modelling platform dependencies (see subsection 7.3.3). "MicroJabberWookie", for example, requires java.lang, java.io and java.util and turned out to be compatible with all Java APIs.

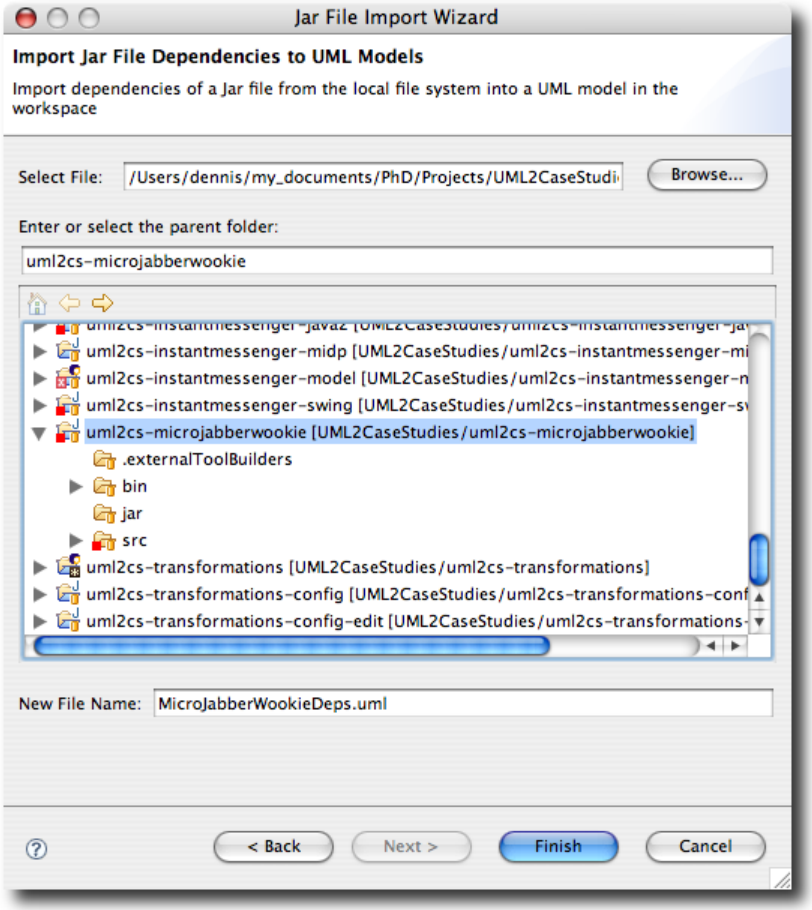

Figure 7.4: Importing the MicroJabberWookie.jar file

7.3.3 Modelling Platform Dependencies

Platform dependencies are modelled for the common base architecture of the SPL as well as for the optional features. Platform dependencies are described in OWL-DL. We use the Protégé ontology editor to model these ontologies.

Before modelling the platform dependencies, it is a good idea to prepare the platform vocabulary ontologies such that the Protégé ontology editor can access them. These ontologies are bundled with the PlatformKit Eclipse Plugin. Open a terminal window or command prompt and issue the following commands (excluding the '$'):

```
$ jar xvf /Applications/eclipse/plugins/be.ac.vub.platformkit_1.1.6.jar ontology
  created: ontology/
  created: ontology/codamos_2005_01/
  created: ontology/codamos_2006_01/
  created: ontology/codamos_2007_01/
  created: ontology/davy_2006_01/
 inflated: ontology/codamos_2005_01/Units.owl
 inflated: ontology/codamos_2006_01/Context.owl
```

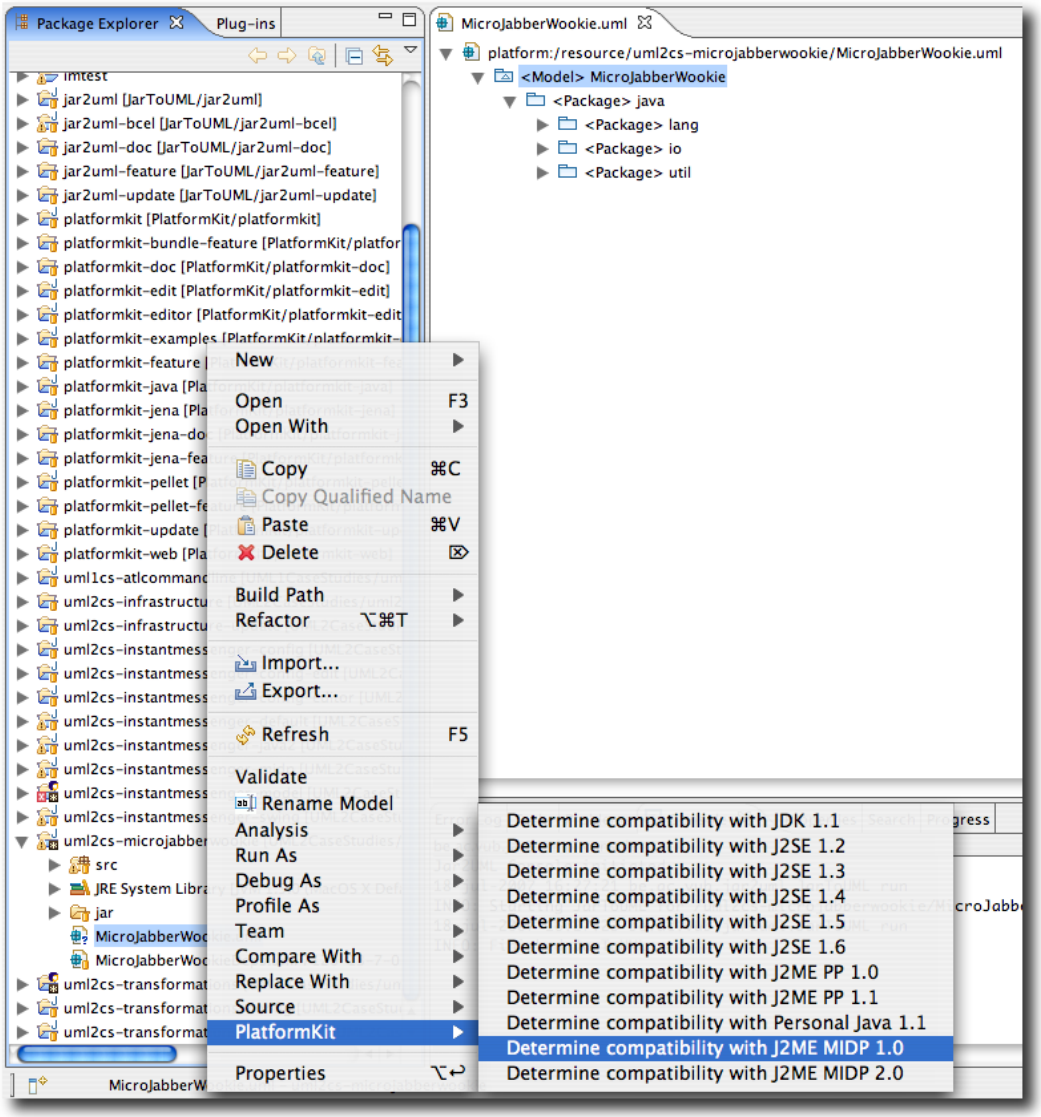

Figure 7.5: Determine compatibility

```
 inflated: ontology/codamos_2006_01/Corba.owl
 inflated: ontology/codamos_2006_01/Environment.owl
 inflated: ontology/codamos_2006_01/Java.owl
 inflated: ontology/codamos_2006_01/OperatingSystems.owl
 inflated: ontology/codamos_2006_01/Platform.owl
 inflated: ontology/codamos_2006_01/Service.owl
 inflated: ontology/codamos_2006_01/User.owl
 inflated: ontology/codamos_2007_01/Java.owl
 inflated: ontology/codamos_2007_01/PackageManagers.owl
 inflated: ontology/codamos_2007_01/Platform.owl
 inflated: ontology/davy_2006_01/Component.owl
 inflated: ontology/davy_2006_01/Draco.owl
$ jar xvf /Applications/eclipse/plugins/be.ac.vub.platformkit.java_1.1.1.jar ontology
  created: ontology/
  created: ontology/codamos_2007_01/
```

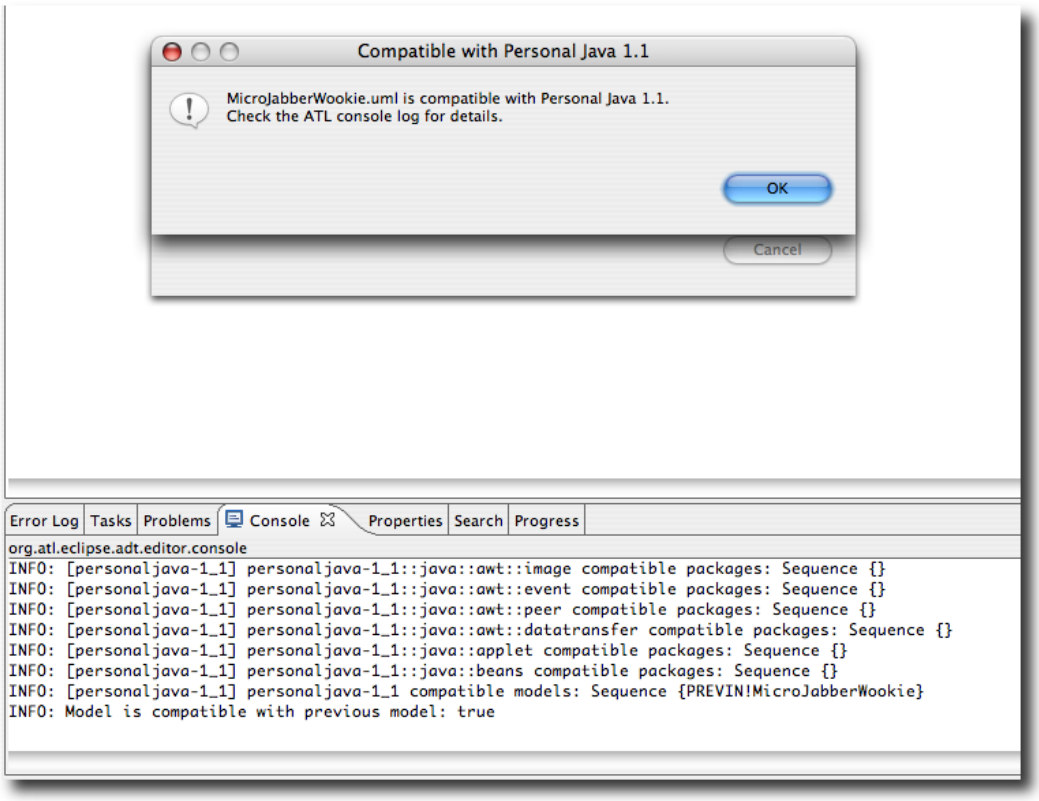

Figure 7.6: Compatibility report

```
inflated: ontology/codamos_2007_01/JavaAPI.owl
inflated: ontology/codamos_2007_01/j2me-midp-1_0-api.owl
inflated: ontology/codamos_2007_01/j2me-midp-2_0-api.owl
inflated: ontology/codamos_2007_01/j2me-pp-1_0-api.owl
inflated: ontology/codamos_2007_01/j2me-pp-1_1-api.owl
inflated: ontology/codamos_2007_01/j2se-1_2-api.owl
inflated: ontology/codamos_2007_01/j2se-1_3-api.owl
inflated: ontology/codamos_2007_01/j2se-1_4-api.owl
inflated: ontology/codamos_2007_01/j2se-1_5-api.owl
inflated: ontology/codamos_2007_01/j2se-1_6-api.owl
inflated: ontology/codamos_2007_01/jdk-1_1-api.owl
inflated: ontology/codamos_2007_01/personaljava-1_1-api.owl
```

The expected output is also listed after the commands (without preceding '$'). We assume here that Eclipse is installed in `/Applications/eclipse` and that you use version 1.1.6 of the `be.ac.vub.platformkit` plugin and version 1.1.1 of the `be.ac.vub.platformkit.java` plugin. Adjust the above commands to reflect your particular Eclipse installation location and plugin versions.

You should now have a folder named "ontology" inside the folder from where you executed the above commands (usually your home directory).

Start up your Protégé editor and do the following:

- Click “New project”.
- Select an “OWL / RDF Files” project and click “Next” (see Fig. 7.7).
- Enter an ontology URI starting with `http://local/` (for example: `http://local/InstantMessenger.owl`) and click “Next” (see Fig. 7.8).
- Select the “OWL DL” language profile and click “Finish” (see Fig. 7.9).

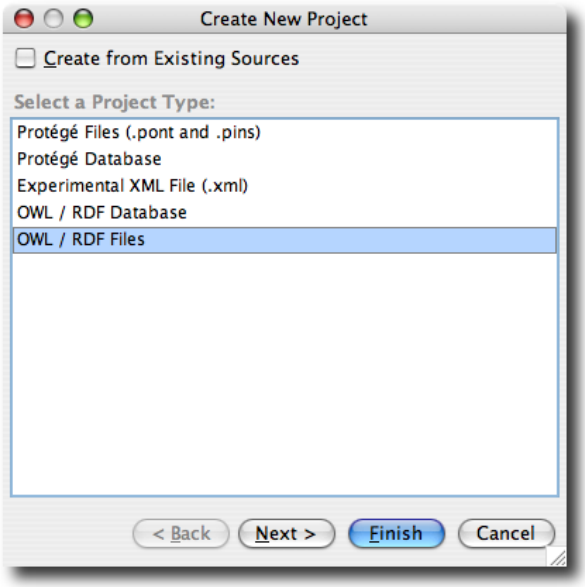

Figure 7.7: New Protégé project kind

You should now see an empty ontology in the main screen of the Protégé editor. In order to build our platform dependency ontology on top of the vocabulary that we prepared earlier, we need to import this vocabulary:

- In the main Protégé screen, click the “Import ontology” button as marked by a red circle in Fig. 7.10.
- Select “Import an ontology contained in one of the available repositories” and click “Next” (see Fig. 7.11).
- Now click “Add repository” as marked by a red oval in Fig. 7.12.
- Select “Local folder” as repository type and click “Next” (see Fig. 7.13).
- Enter the folder name of the vocabulary that we prepared earlier, select “Include sub-folders” and click “Finish” (see Fig. 7.14).

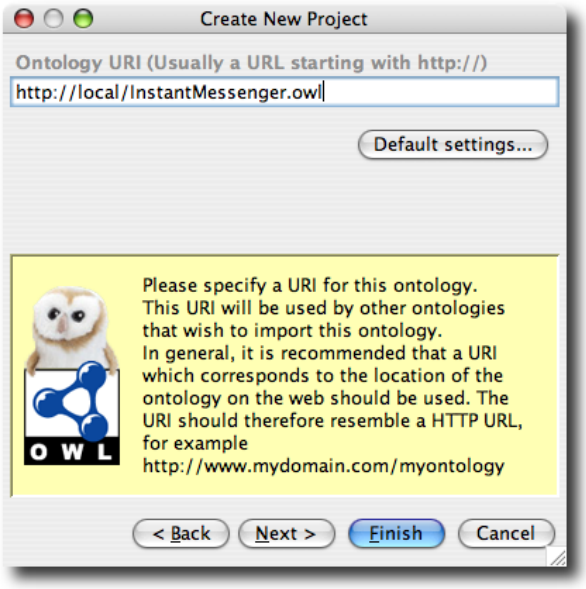

Figure 7.8: New Protégé project URI

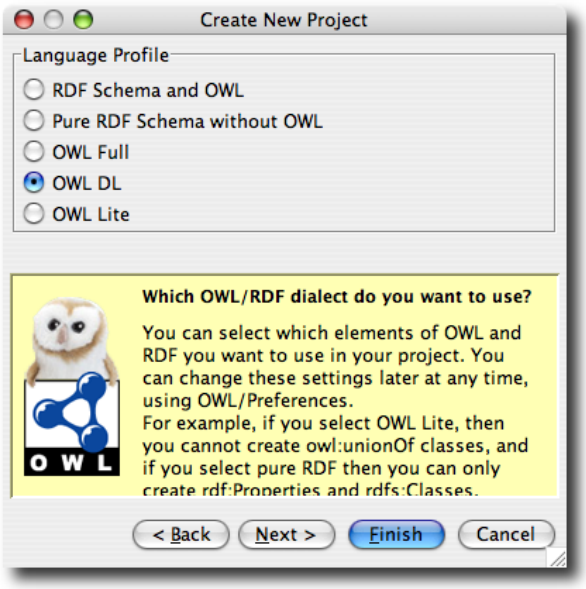

Figure 7.9: New Protégé project language profile

- Now select the http://ssel.vub.ac.be/codamos/2007/01/JavaAPI.owl ontology from the list and click "Next" (see Fig. 7.15).
- In the next screen, click "Finish".
- Review the prefixes of the imported ontologies (see Fig. 7.16). You may

want to change the prefix of the http://ssel.vub.ac.be/codamos/2007/01/JavaAPI.owl ontology to `java-api`. Click "Close" when you're done.

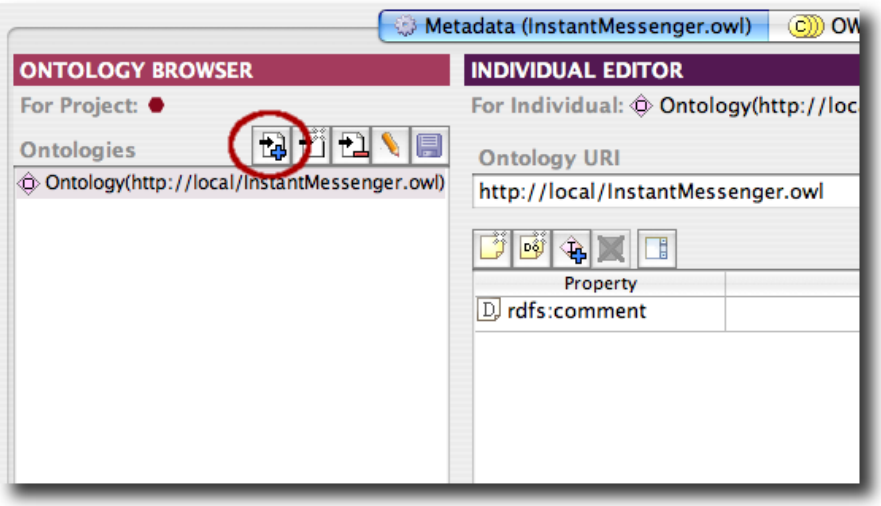

Figure 7.10: Importing an ontology into Protégé

Now that the empty ontology is set up, we can add platform dependencies. It is highly recommendable to use a DIG reasoner while modelling your platform dependencies. The DIG reasoner URL can be set in Protégé via the "OWL → Preferences" menu. As a DIG reasoner, you can use Racer, Pellet or Fact. You can use the DIG reasoner to classify the taxonomy based on the ontology facts. This is done via "OWL → Classify taxonomy".

Now we can add platform dependency classes to the ontology. We assume that you can find your way around Protégé a bit, such that you can do basic ontology editing. Please save your ontology often during the following steps, as we may reach the memory limits of Protégé. This is the guideline procedure for adding platform dependencies:

- A platform dependency is always a subclass of `platform:Platform`. Start by adding a new subclass to `platform:Platform`. Name this class `InstantMessengerPlatform`. This class will represent the platform dependency constraint for the common instant messenger architecture.
- Depending on the platform features we want to require, we need to define extra `platform:Feature` subclasses. InstantMessengerPlatform will require a specific kind of `java:JRE`. Add a subclass to `java:JRE` and name it `BasicJRE`.

A `java:JRE` typically provides a built-in `java:JavaLibrary`. Use the DIG reasoner taxonomy classification result to find out which `java:JavaLibrary`

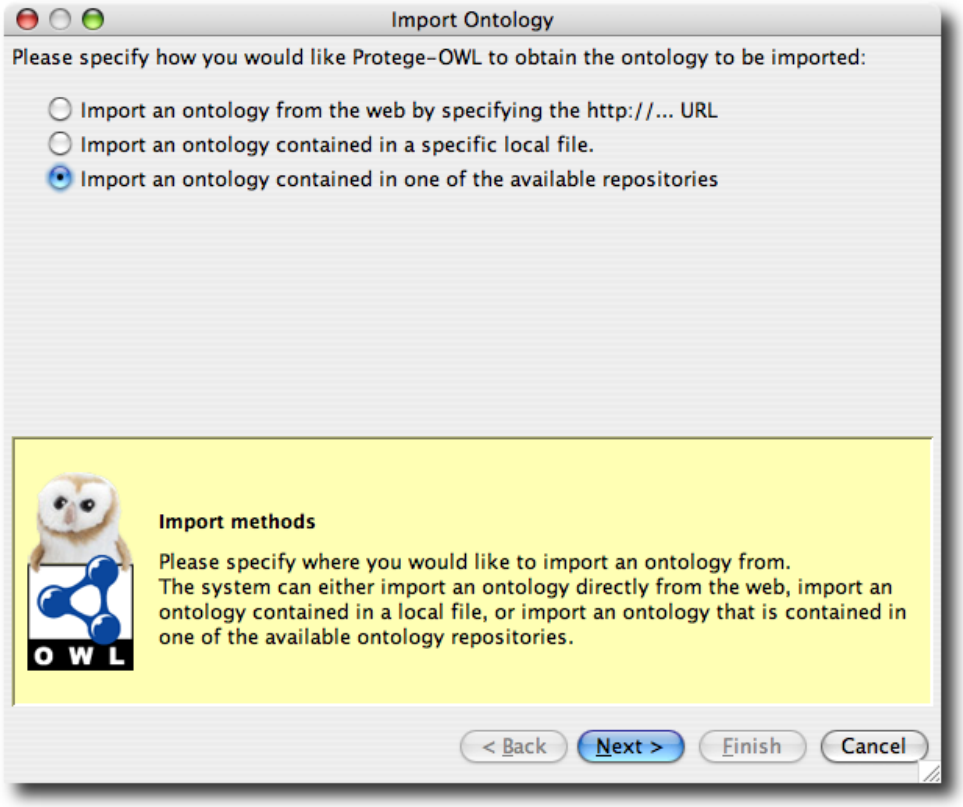

Figure 7.11: Import an ontology from a repository

subclasses we require. For `BasicJRE` we require a basic version of `java.lang` and `java.util`. The Java class library that provides the `java.lang` package and the `java.util` package are represented by (several versions of) the classes `JavaLangLibrary` and `JavaUtilLibrary`, respectively. `j2me-midp-1_0:JavaLangLibrary` is sufficient for our requirements and is subclassed by most other versions of `JavaLangLibrary`, i.e. most other versions of `JavaLangLibrary` are compatible with `j2me-midp-1_0:JavaLangLibrary`. `personaljava-1_1:JavaLangLibrary` is not compatible with `j2me-midp-1_0:JavaLangLibrary`, but is still sufficient for our requirements.

- Add a *necessary & sufficient* restriction on the `java:providesBuiltinJavaLibrary` property to the `BasicJRE` class. As a "Filler", enter the union of all `JavaLangLibrary` classes that are sufficient for our requirements. Any `JavaLangLibrary` classes that are compatible with one of the other classes in the union may be omitted (see Fig. 7.17).

- Add another *necessary & sufficient* restriction on the `java:providesBuiltinJavaLibrary` property to the `BasicJRE` class. As a "Filler",

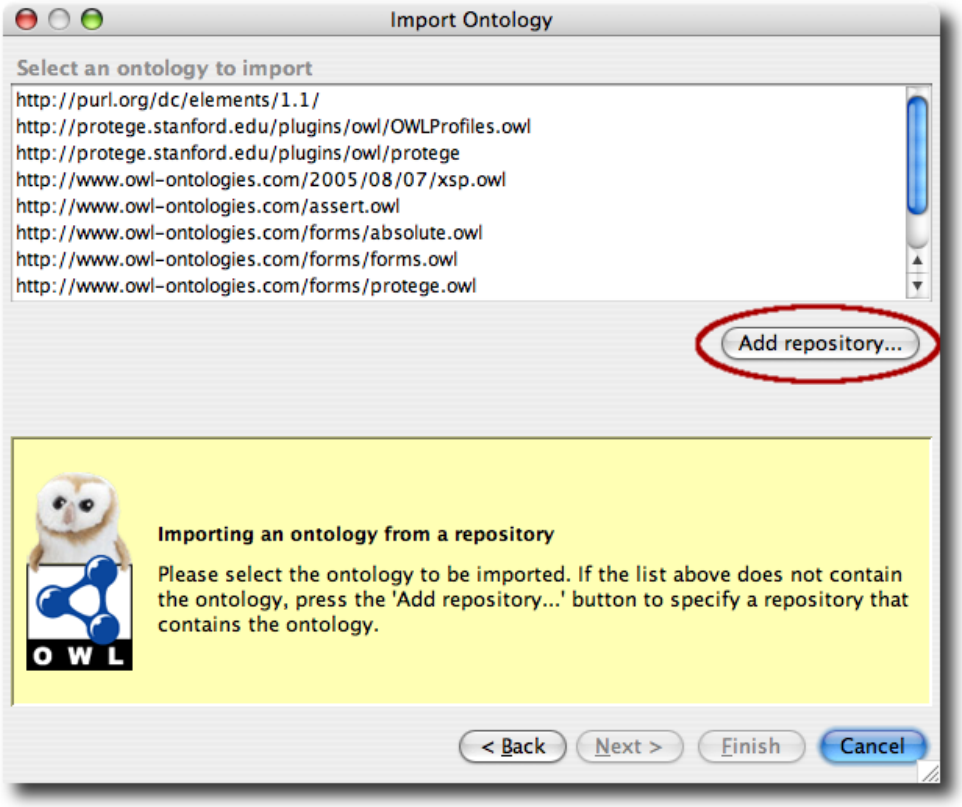

Figure 7.12: Add a new ontology repository

enter the union of `j2me-midp-1_0:JavaUtilLibrary` and `personaljava-1_1:JavaUtilLibrary`. The resulting `BasicJRE` description should look like Fig. 7.18.

- Now add a *necessary & sufficient* restriction on the `platform:providesFeature` property to the `InstantMessengerPlatform` class. As a "Filler", enter the `BasicJRE` class (see Fig. 7.19).

We now have a basic platform dependency class for any instant messenger product. You can add more platform dependency classes for each of the optional instant messenger features, such as "JabberNetwork", "DefaultJabberTransport", "AWTUserInterface", etc. It is also possible to require a particular package manager: see the `platform:PackageManager` class, the `java:JavaPackageManager` class and the `java:providesJavaPackageManager` property.

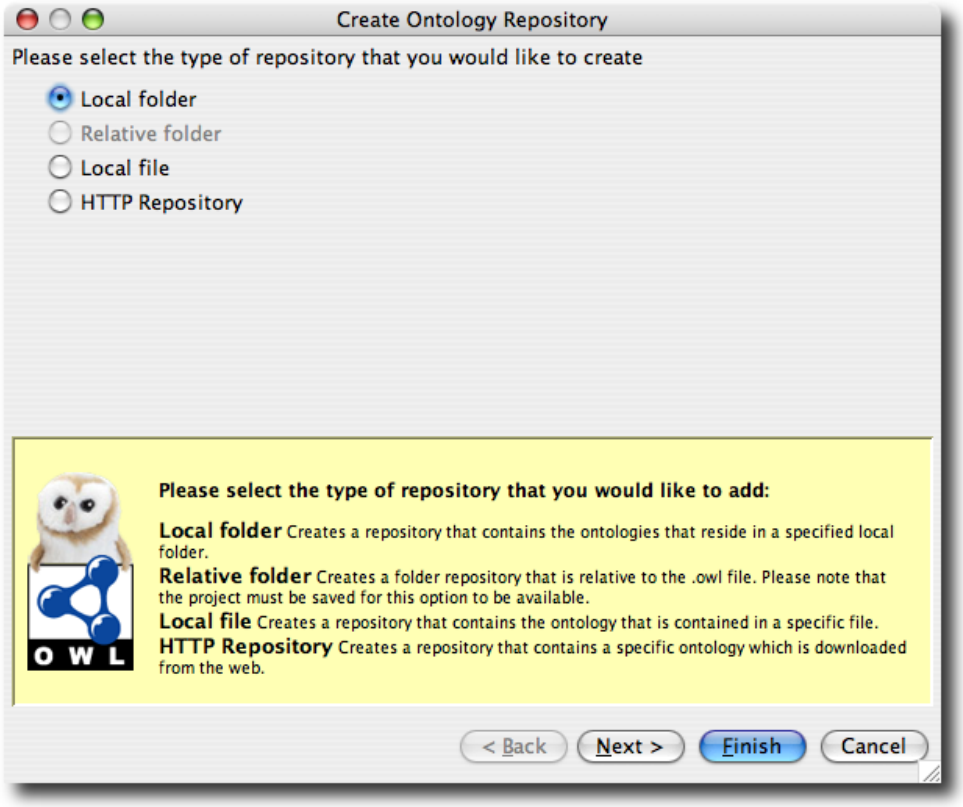

Figure 7.13: Select repository type

After you've added all platform dependency classes, you can get an idea of which is the most specific platform dependency by choosing "OWL → Classify taxonomy" again. The leaf classes in the hierarchy are the most-specific platform requirements. Note that this step will also be performed by PlatformKit itself.

7.3.4 Setting up a Platform-Aware Configuration Language

We have set up the core assets of our SPL and we have modelled their platform dependencies in an ontology. Now we define a domain-specific modelling language for the configuration of our SPL, in which the platform dependencies and the core assets are brought together. The meta-model of this configuration language is annotated with the names of our platform dependency classes, such that PlatformKit can extract them for each meta-class in the configuration language. The goal is to enable PlatformKit to profile configuration models

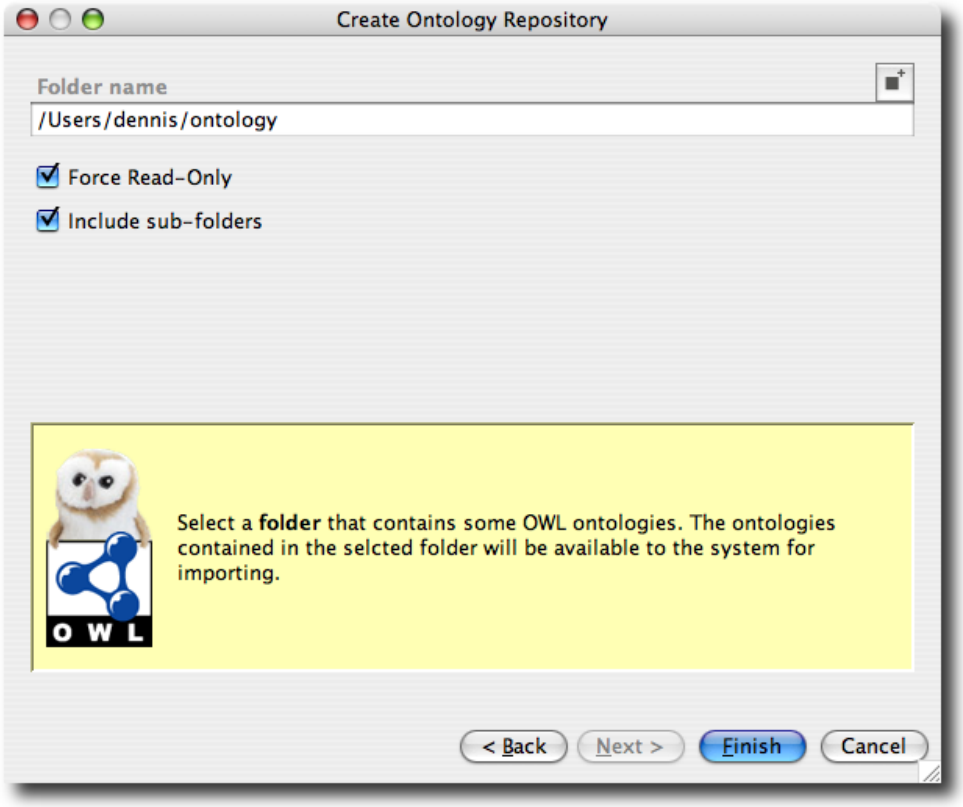

Figure 7.14: Create new repository

against a platform instance description and see whether a particular configuration works for a given platform. A flowchart of the entire procedure is shown in Fig. 6.5 in chapter 6, subsection 6.4.1. Note that the task of PlatformKit profiling itself is discussed in subsection 7.3.5.

The first step in creating our configuration language is to define an EMF meta-model. We assume basic experience with EMF meta-modelling and refer to the EMF documentation[17] for instructions. As an example, we will use the configuration language meta-model of the Instant Messenger case study, as shown in Fig. 6.3 in chapter 6, subsection 6.3.2.

We now describe the steps to add the platform dependencies to this meta-model. You can use the EMF Ecore editor to do this:

- Add an EAnnotation to the main EPackage ("instantmessenger" for our example) and set its "Source" property to "PlatformKit".

- Add a Details Entry to the "PlatformKit" EAnnotation.

[17] http://www.eclipse.org/modeling/emf/docs/

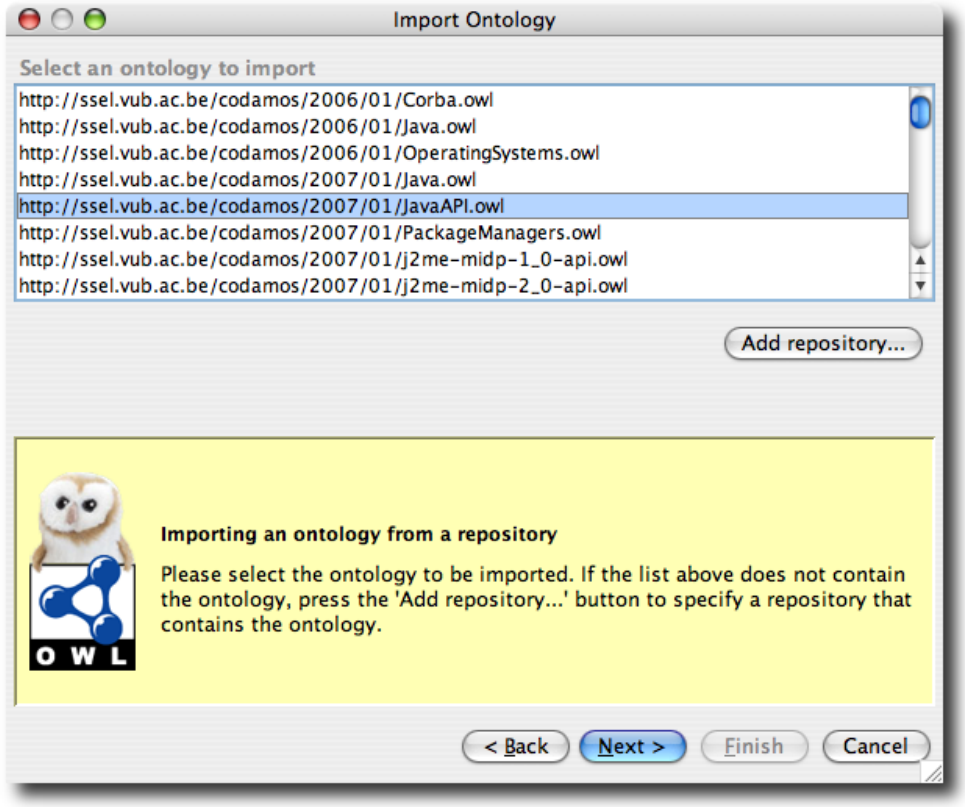

Figure 7.15: Select ontology to import

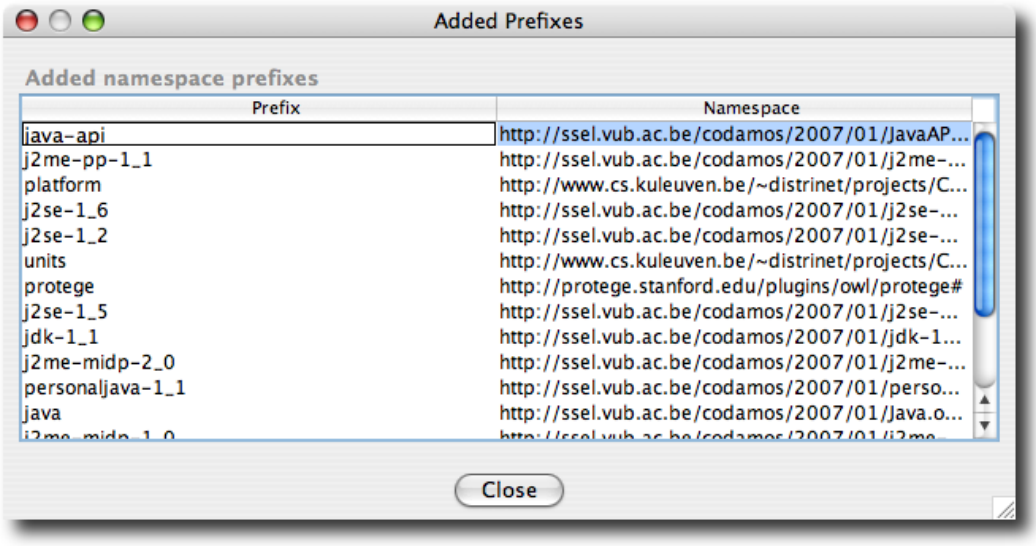

Figure 7.16: Review the added ontology prefixes

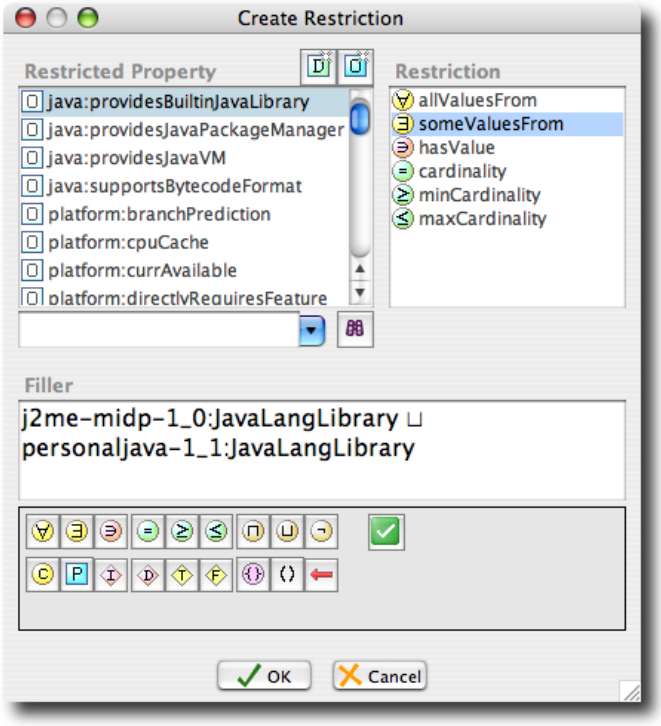

Figure 7.17: Adding a restriction to a JRE

- Set the "Key" property of the Details Entry to "Ontology" and the "Value" property to "InstantMessenger.owl" (or the name you chose for your platform dependency ontology).
- For each meta-class that corresponds to a platform dependency, add an EAnnotation and set its "Source" property to "PlatformKit".
- Add a Details Entry to this EAnnotation with its "Key" property set to "PlatformConstraint" and its "Value" property set to the URL of its platform dependency class. This URL takes the form `<ontology URL>#<class name>`. The "InstantMessengerConfiguration" meta-class, for example, has its "PlatformConstraint" value set to "`http://local/InstantMessenger.owl#InstantMessengerPlatform`".

The resulting meta-model should look like Fig. 7.20 in the EMF Ecore editor.

Once the meta-model has been defined and annotated, a Platformkit Model must be created:

- Choose "File → New → Other".
- Select "PlatformKit → Platformkit Model" and click "Next".

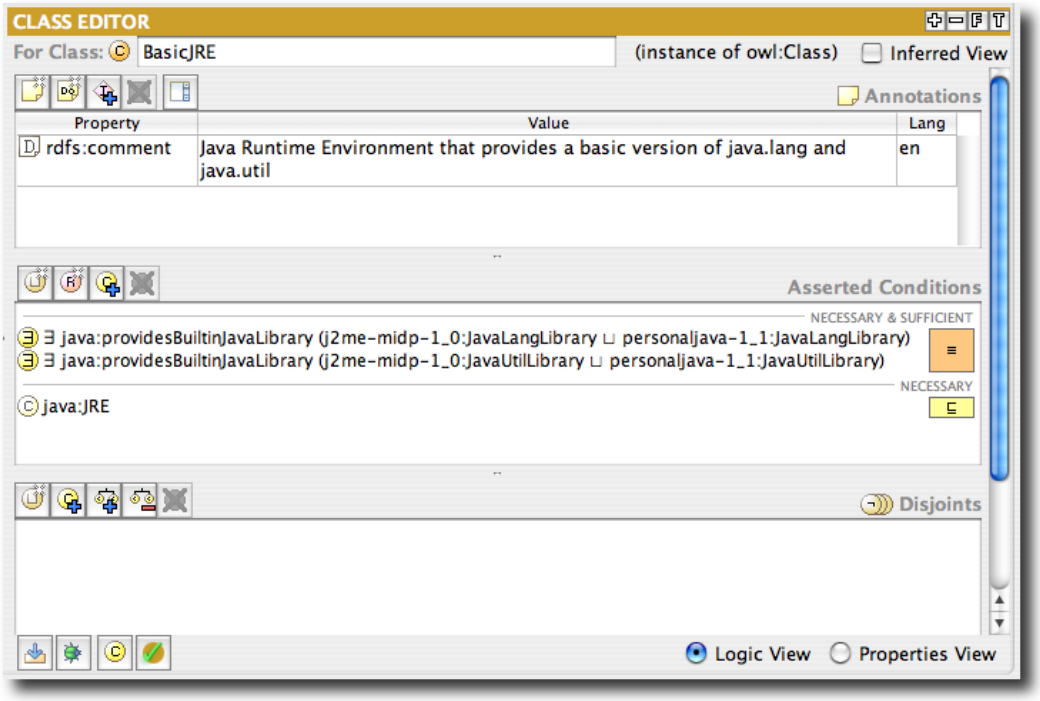

Figure 7.18: Description of BasicJRE

- Select the folder that contains the Ecore meta-model as parent folder and use the same base name as file name. For example, use "InstantMessenger.platformkit" if your meta-model is saved as "InstantMessenger .ecore".

- Click "Finish".

You now have an empty PlatformKit Model with a root "ConstraintSpace" element:

- Right-click the "ConstraintSpace" element and select "Add Product Line Meta-model" as shown in Fig. 7.21.

- Select the Ecore meta-model of the configuration language, including all referenced meta-models, then click "OK". For example, "InstantMessenger.ecore" references "Transformations.ecore", which means both have to be included as shown in Fig. 7.22.

Your Platformkit Model now has several "ConstraintSet" elements in its "ConstraintSpace". We will now pre-sort the Platformkit Model *most-specific-first*:

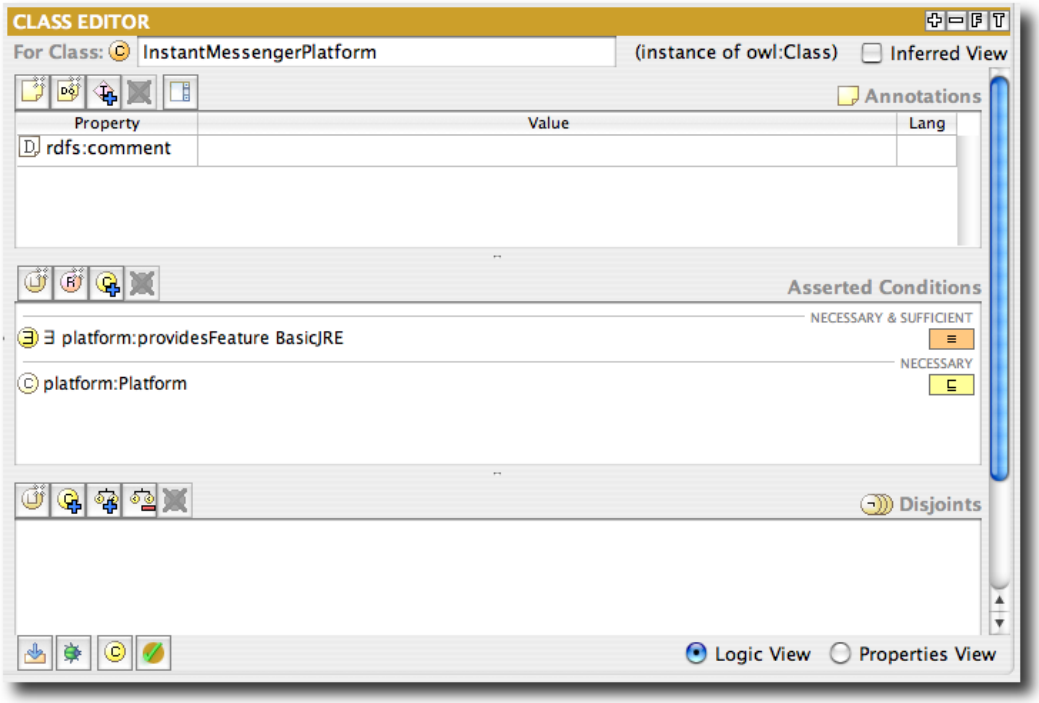

Figure 7.19: Description of InstantMessengerPlatform

- Right-click the "ConstraintSpace" element and select "Classify Taxonomy" as shown in Fig. 7.23. The classification procedure may take several minutes.

- As a result, you should have a ".inferred.owl" file with the same base name as the Platformkit Model, which is also stored in the same folder. For example, you should have an "InstantMessenger.inferred.owl" file for the "InstantMessenger.platformkit" model.

- Right-click the "ConstraintSpace" element again and select "Sort Most Specific First".

- As a result, the PlatformKit Model will be re-sorted with the most-specific "ConstraintSet" element at the top.

- Save the resulting Platformkit Model.

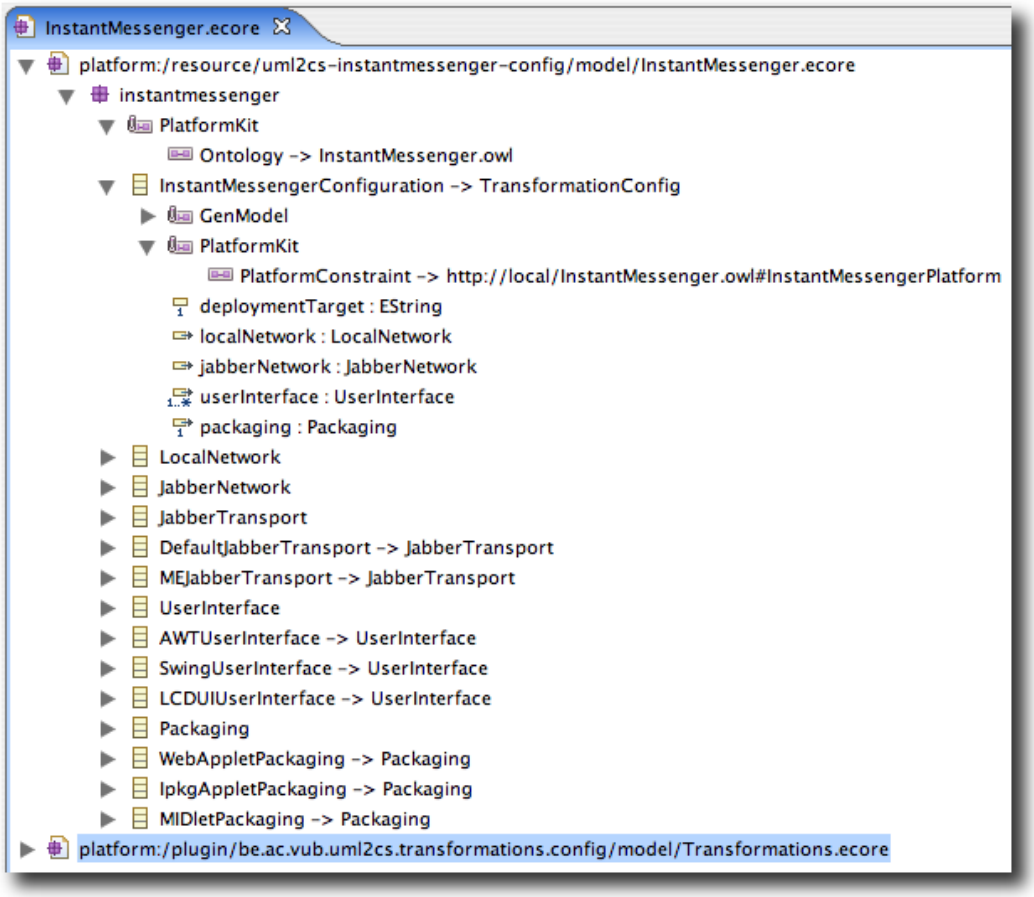

Figure 7.20: Compatibility report

> Normally, the built-in Pellet OWL DL reasoner is used to perform the above steps. It is also possible to use an external OWL DL reasoner via the DIG interface. This is done by selecting "Window → Preferences" from the menu and changing the settings in the "PlatformKit" category.

In case you wish to package the configuration language as an EMF plugin, you need to bundle the Platformkit Model and the ontologies:

- Include all ".platformkit", ".owl" and ".inferred.owl" files in your "build .properties" and make sure they are in the same folder as the ".ecore" file.
- Create extra URI mappings for all included ".platformkit", ".owl" and ".inferred.owl" files. Fig. 7.24 shows the extra URI mappings for the

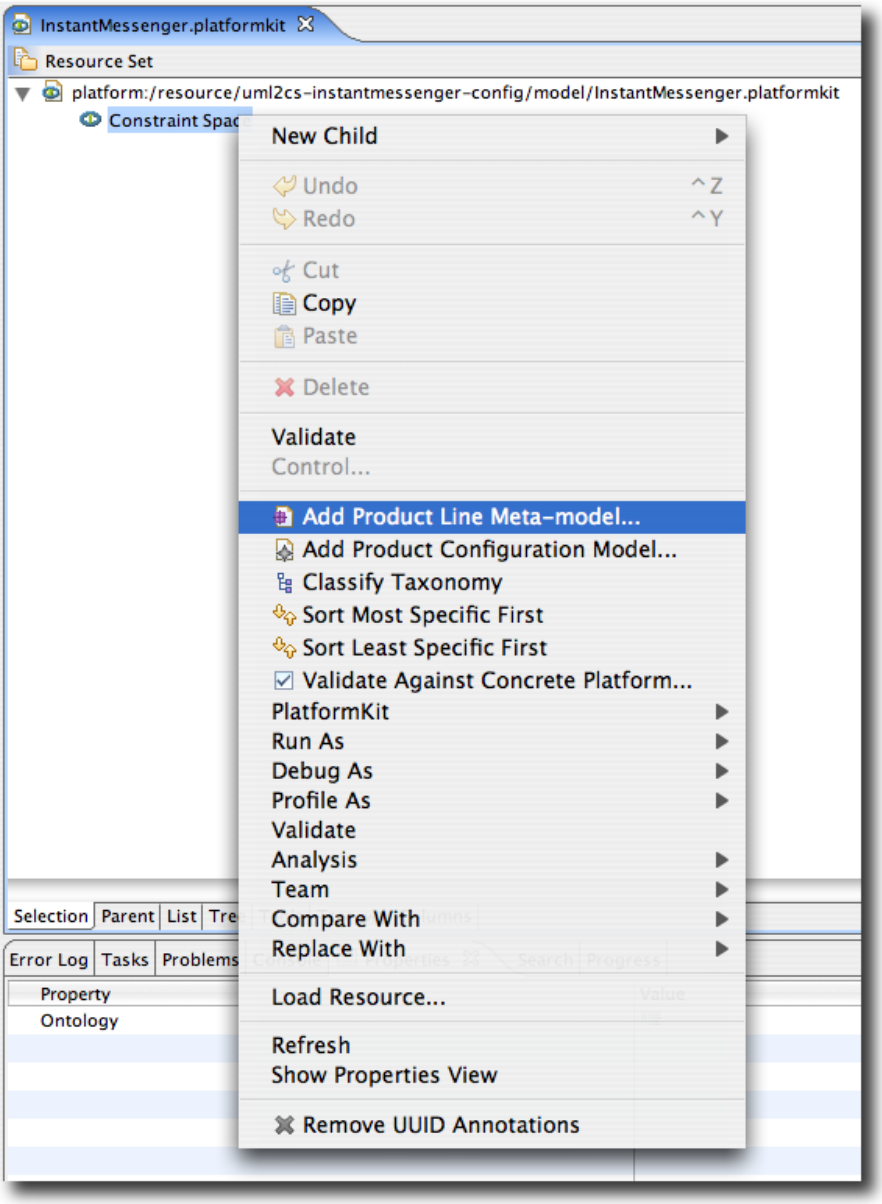

Figure 7.21: Adding a product line meta-model

instant messenger configuration language plugin.

7.3.5 Platform-Driven Configuration

Each configuration model that is expressed in a platform-aware configuration language can be checked by PlatformKit to see for which platforms it is valid.

When editing a product configuration model in the tree-based EMF editor,

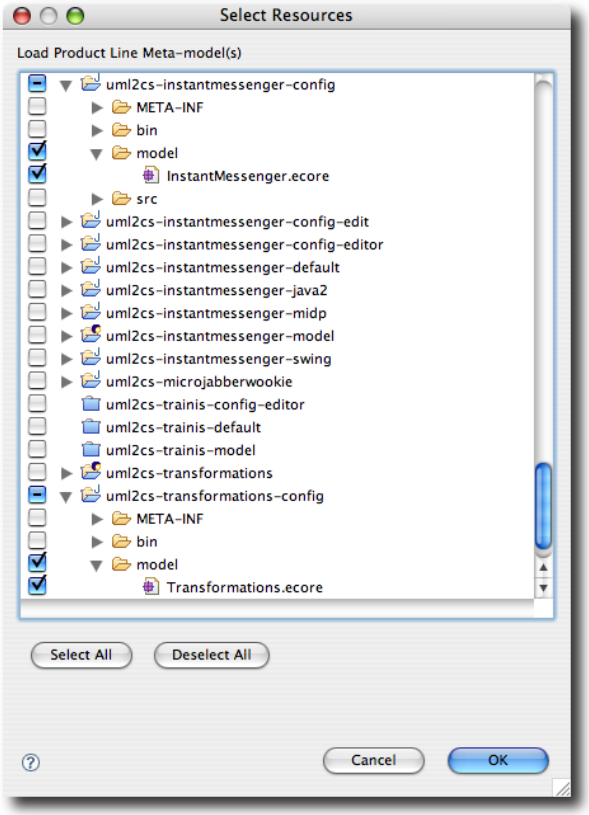

Figure 7.22: Selecting product line meta-models

one can add new child elements to the model. For the instant messenger configuration language, the options are listed in Fig. 7.25. EMF already takes care of multiplicity constraints given in the meta-model by greying out the invalid options. PlatformKit can further limit the available options by removing the options that are invalid for a chosen platform:

- Right-click the root element of the configuration model and select "PlatformKit → Profile Against Concrete Platform", as shown in Fig. 7.26.

- Select a platform instance description from the list of built-in platforms and click "OK", as shown in Fig. 7.27. Alternatively, select "Platform specification from file" after which you can provide a platform instance specification ".owl" file from the workspace. Example platform instance specifications can be found at http://ssel.vub.ac.be/viewvc/PlatformKit/platformkit-examples/codamos_2007_01/.

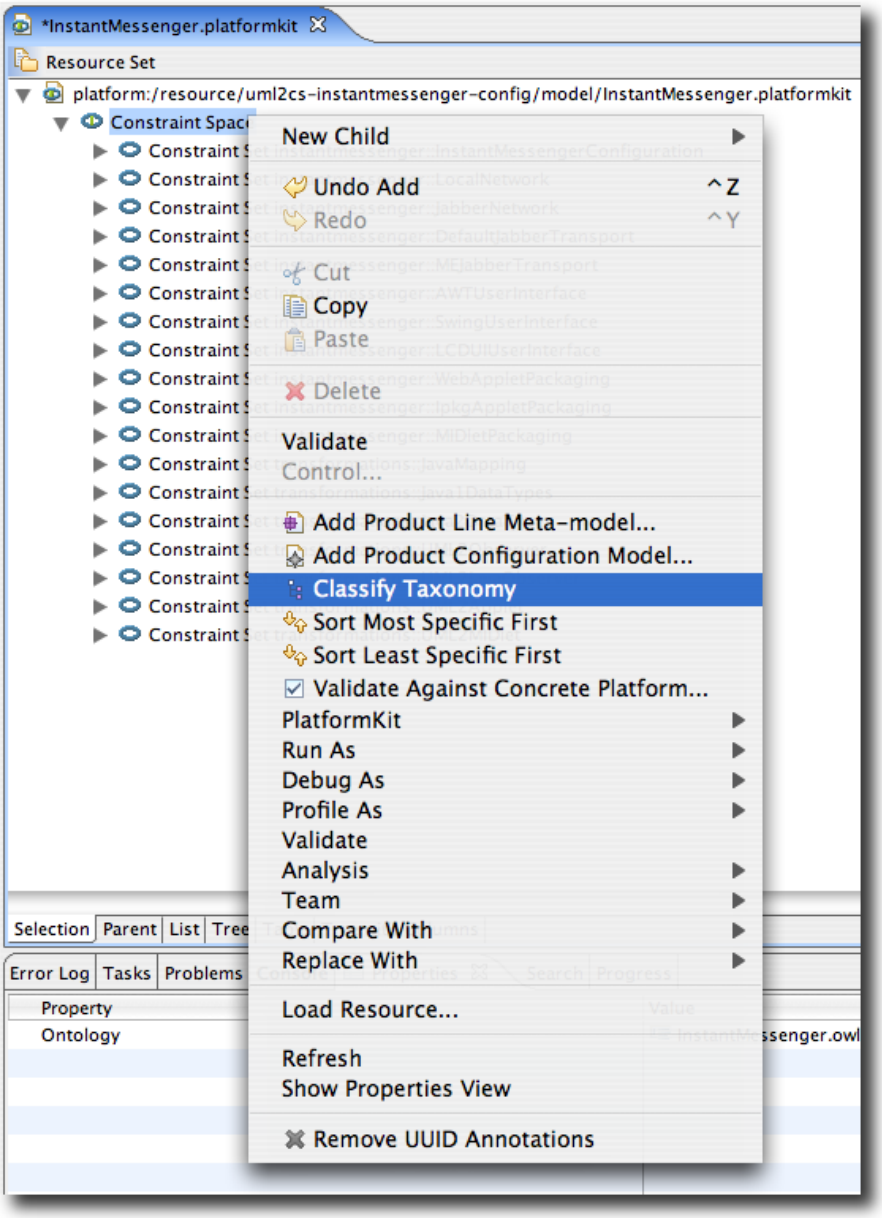

Figure 7.23: Classify taxonomy

Normally, the built-in Pellet OWL DL reasoner is used to perform the above steps. It is also possible to use an external OWL DL reasoner via the DIG interface. This is done by selecting "Window → Preferences" from the menu and changing the settings in the "PlatformKit" category.

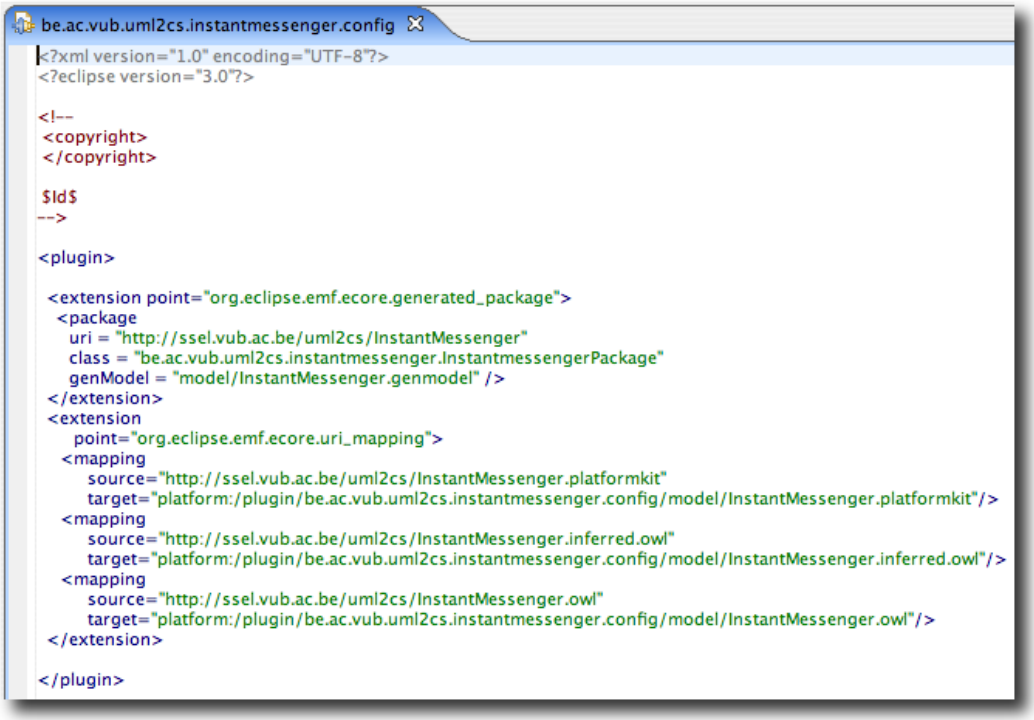

Figure 7.24: Extra URI mappings for EMF plugins

As a result, the amount of configuration options has been reduced to valid options only. Fig. 7.28 shows the valid options for configuring an instant messenger for a JDK 1.1 PC platform. Any existing configuration choices can be validated against the chosen platform instance:

- Right-click the root element of the configuration model and select "Validate".

If there are any invalid elements in the configuration model, they will be reported, as shown in Fig. 7.29.

7.3.6 Platform-Driven Deployment

Each configuration model that is expressed in a platform-aware configuration language can be checked by PlatformKit to see for which platforms it is valid. In addition, all product configuration models in a product line can be compared against eachother to see which product configurations are more platform-specific than others. The PlatformKit Servlet can eventually deploy the product configuration of choice. A flowchart of setting up platform-driven deployment is shown in Fig. 6.7 in chapter 6, subsection 6.4.2.

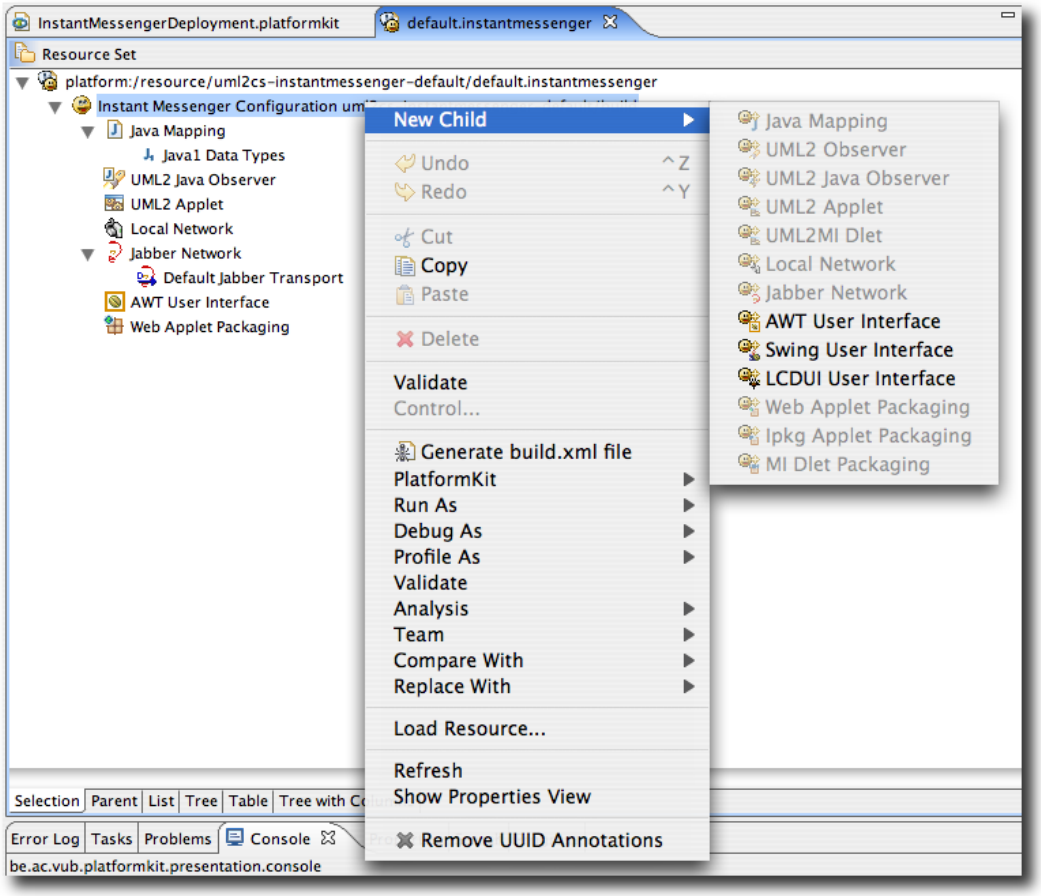

Figure 7.25: New child options for instant messenger configuration

In the previous task, we have created a Platformkit Model for the platform-aware configuration language. Platformkit Models can also be used to support deployment. Instead of a product line's meta-classes, it will contain representations of the product configurations. First, we create a new Platformkit Model:

- Choose "File → New → Other".
- Select "PlatformKit → Platformkit Model" and click "Next".
- Select a folder. This folder is meant to be deployed on a website later.
- Click "Finish".

You now have an empty PlatformKit Model with a root "ConstraintSpace" element:

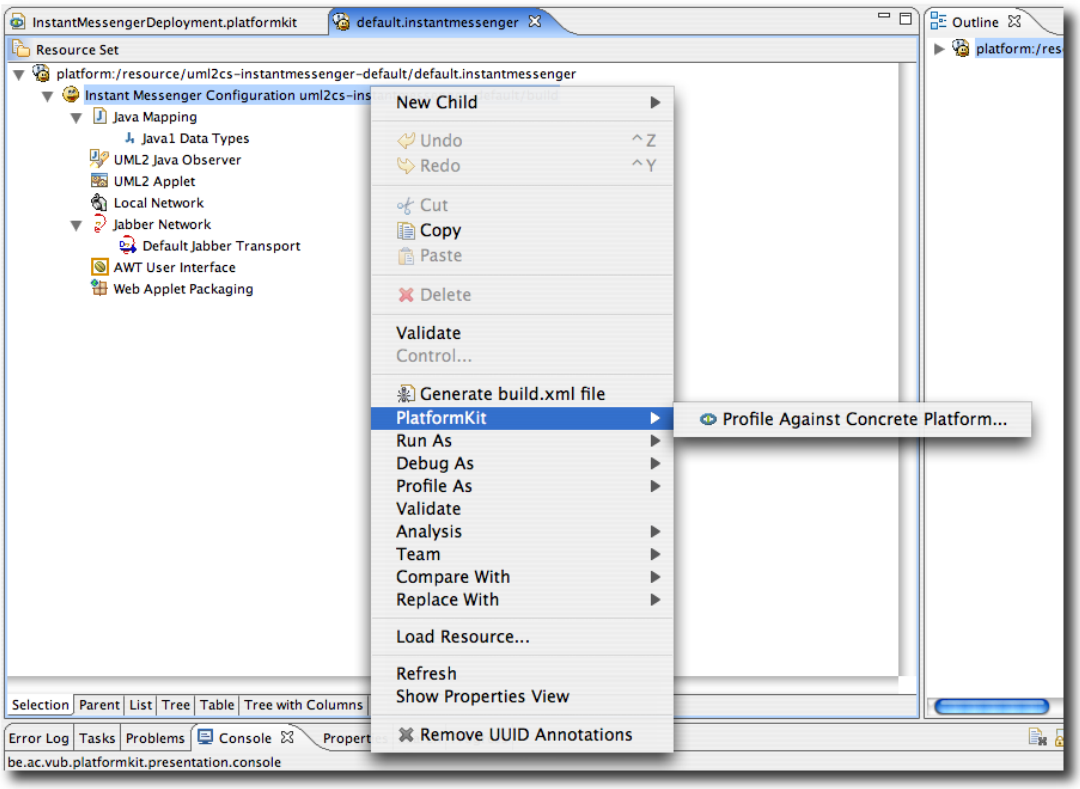

Figure 7.26: Profile against concrete platform

- Right-click the "ConstraintSpace" element and select "Add Product Configuration Model" as shown in Fig. 7.30.
- Select the configuration models of all configurations that you want to include in deployment, then click "OK". For the instant messenger example, this is shown in Fig. 7.31.

Your Platformkit Model now has several "ConstraintSet" elements in its "ConstraintSpace". Note that the names of each "ConstraintSet" will be used later by the PlatformKit Servlet as a relative URL to which clients are redirected for deployment. The name of each "ConstraintSet" must therefore point to a download location of the corresponding product configuration. For the instant messenger product line, the "ConstraintSet" names point either to downloadable software packages or to a web page that will display an applet (see Fig. 7.32).

We will now pre-sort the Platformkit Model *most-specific-first*:

- Right-click the "ConstraintSpace" element and select "Classify Taxonomy" as shown in Fig. 7.33. The classification procedure may take several minutes.

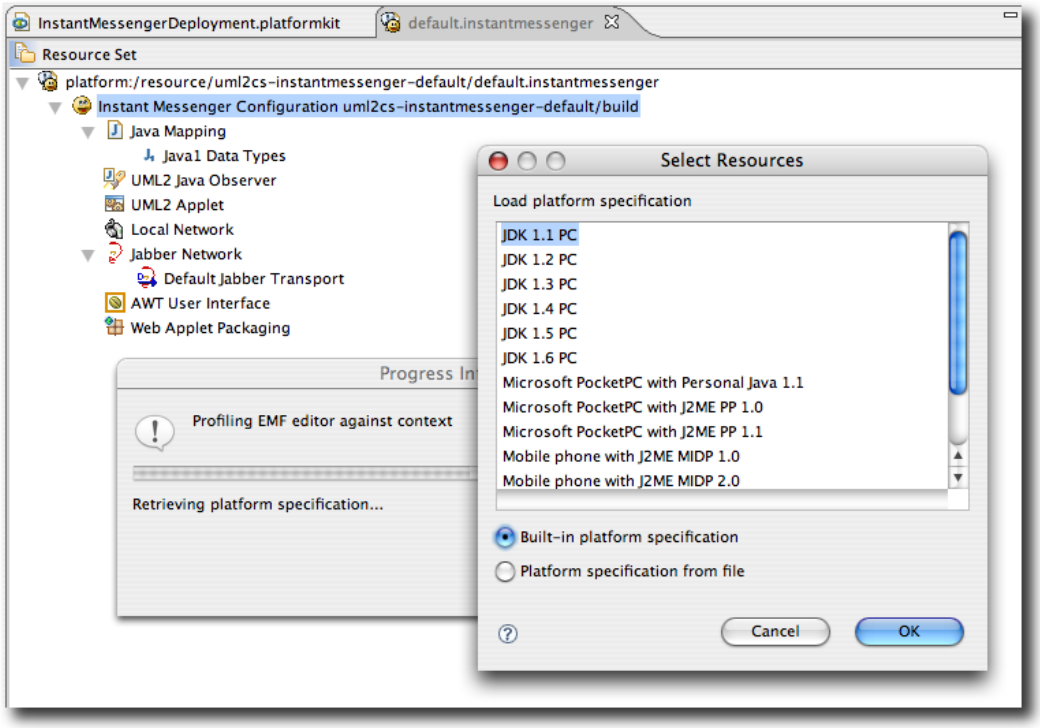

Figure 7.27: Select a platform specification to profile against

- As a result, you should have a ".inferred.owl" file with the same base name as the Platformkit Model, which is also stored in the same folder. For example, you should have an "InstantMessengerDeployment.inferred .owl" file for the "InstantMessengerDeployment.platformkit" model.
- Right-click the "ConstraintSpace" element again and select "Sort Most Specific First".
- As a result, the PlatformKit Model will be re-sorted with the most-specific "ConstraintSet" element at the top.
- Save the resulting Platformkit Model.

Your PlatformKit Model should look like the one shown in Fig. 7.32. The Platformkit Model of all product configurations can be validated against platform instance specifications, similar to Platform-Driven Configuration, where a single product configuration is validated against a platform instance. This way, you can test whether you get the expected result of valid and invalid configurations for each platform instance you validate against. Validation of the Platformkit Model is done as follows:

- Right-click the "ConstraintSpace" element and select "Validate Against Concrete Platform" (see also Fig. 7.33).

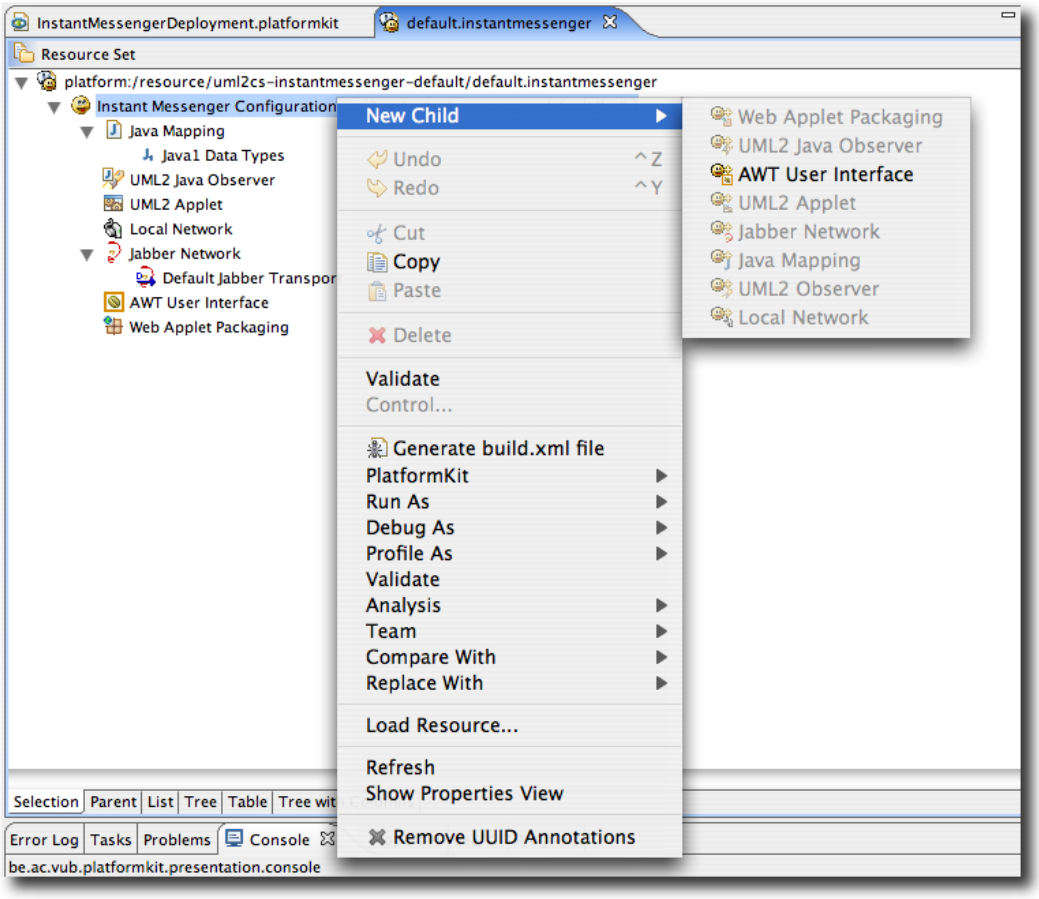

Figure 7.28: Profiled new child options for instant messenger configuration

- Select a platform instance description from the list of built-in platforms and click "OK", as shown in Fig. 7.34. Alternatively, select "Platform specification from file" after which you can provide a platform instance specification ".owl" file from the workspace. Example platform instance specifications can be found at http://ssel.vub.ac.be/viewvc/PlatformKit/platformkit-examples/codamos_2007_01/.

As a result, a dialogue will be shown on the screen with the valid Constraint Sets, as shown in Fig. 7.35. In addition, a ".valid.txt" file will be written next to the Platformkit Model, using the same base name (e.g. "InstantMessenger-Configuration.valid.txt").

Now that the Platformkit Model and its inferred ontology have been prepared, you need to make sure that the deployable products are put in the right place:

- For each "ConstraintSet" element in the Platformkit Model, make sure that there exists a file or folder that corresponds to the name of the

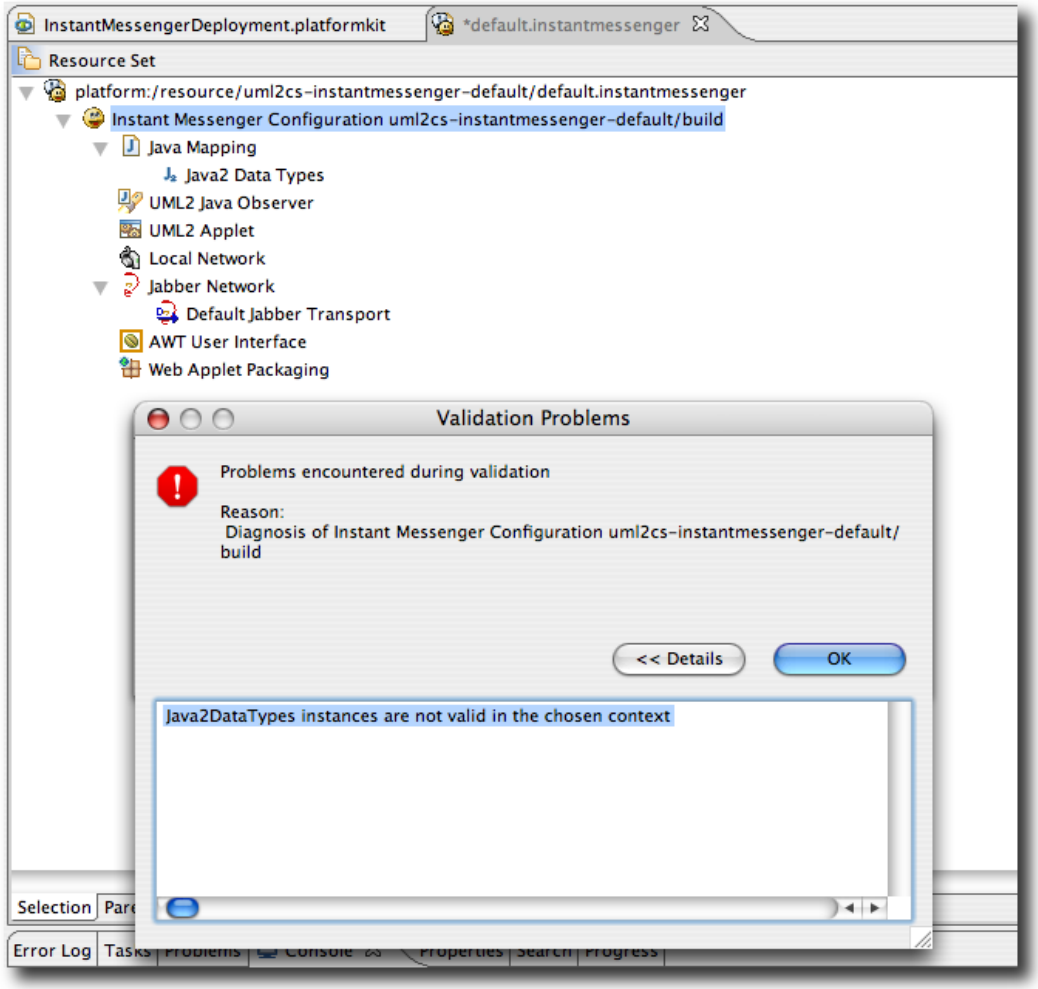

Figure 7.29: Validation result of an instant messenger configuration

"ConstraintSet" element. For example, if the name of a "ConstraintSet" element is "default/applet/", then there must be a path with that name relative to the ".platformkit" file.

- Create a "none" folder next to the ".platformkit" file. If the PlatformKit Servlet finds no suitable product configuration to deploy, clients are redirected here. If you want, you can create an index page inside this folder. An example index page can be found in appendix C.

For the instant messenger product line, the deployment folder is shown in Fig. 7.36. Note that the instant messenger example also employs a special index page in the root of the deployment folder to make invocation of the PlatformKit Servlet easier. Once your deployment folder is complete:

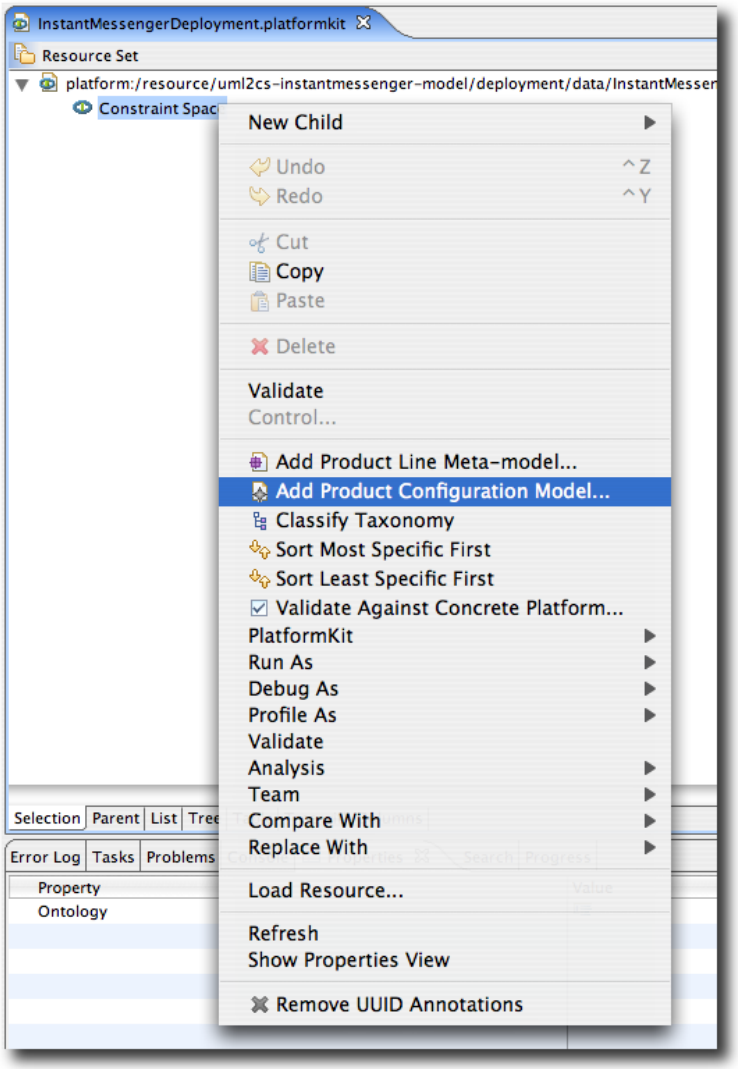

Figure 7.30: Adding a product configuration model

- Upload the folder containing the ".platformkit" and ".inferred.owl" files to a web site, including all its subfolders that contain the deployable product configurations. For the instant messenger example, the contents of the "data" folder are uploaded to a website.

This is it! Now you can use the PlatformKit Servlet to select a product configuration based on a platform description.

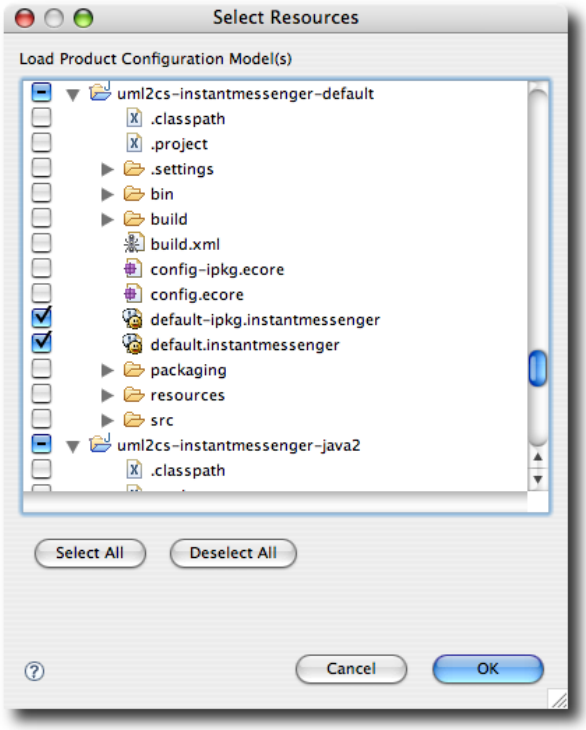

Figure 7.31: Selecting product line configuration models

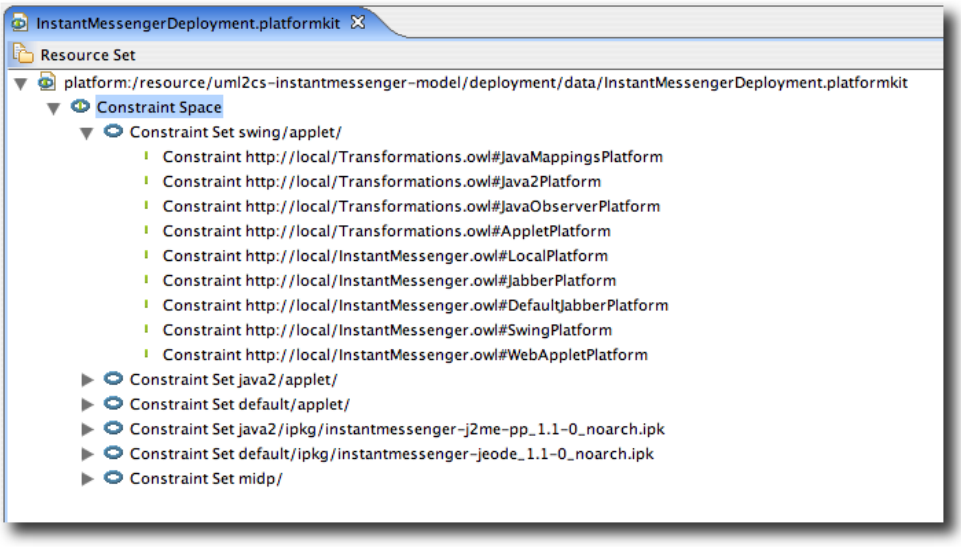

Figure 7.32: The ordered Platformkit Model for instant messenger deployment

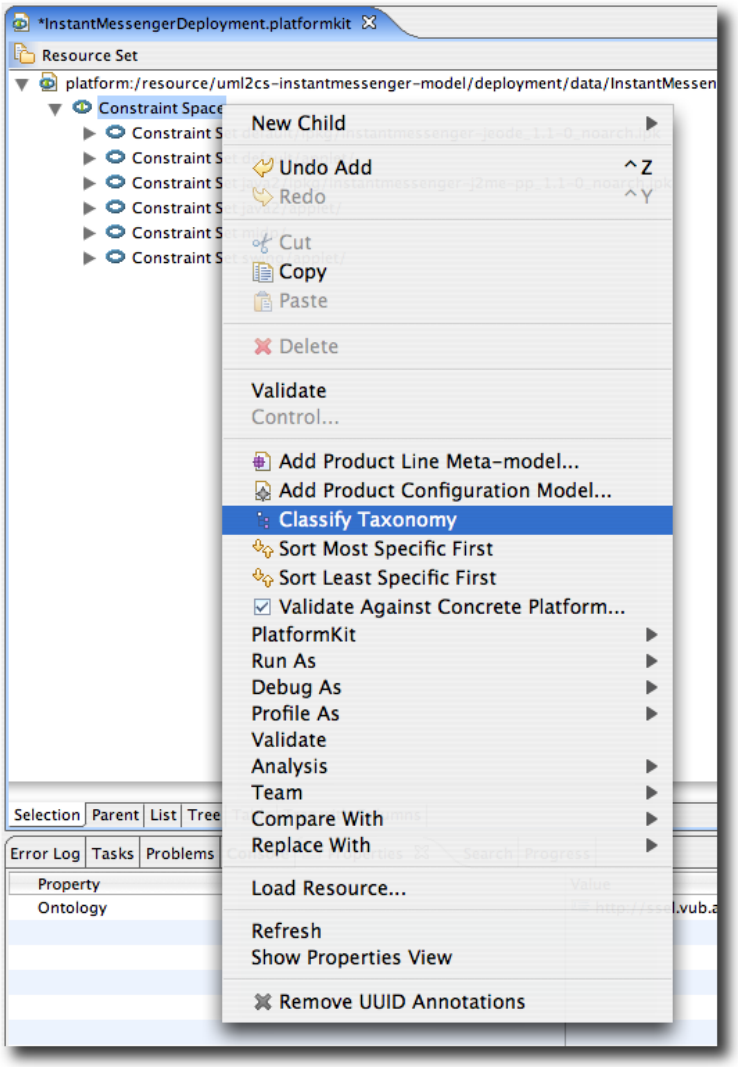

Figure 7.33: Classify taxonomy

7.4 ATL

The ATLAS Transformation Language (ATL) tool has been extended for the purpose of our instant messenger case study. Apart from necessary bugfixes, a number of features have been added to ATL. These features are discussed in the following subsections.

7.4.1 Superimposition

ATL superimposition of transformation modules has been described in chapter 2. It allows the developer to separate the rules of a transformation into

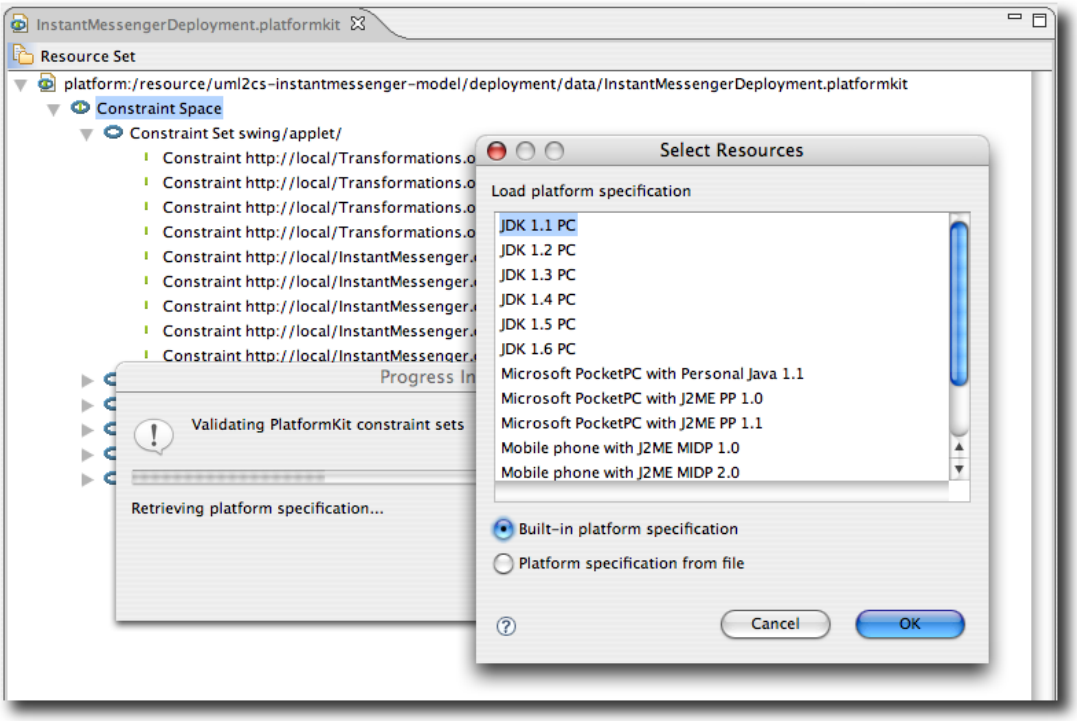

Figure 7.34: Select a platform specification to validate against

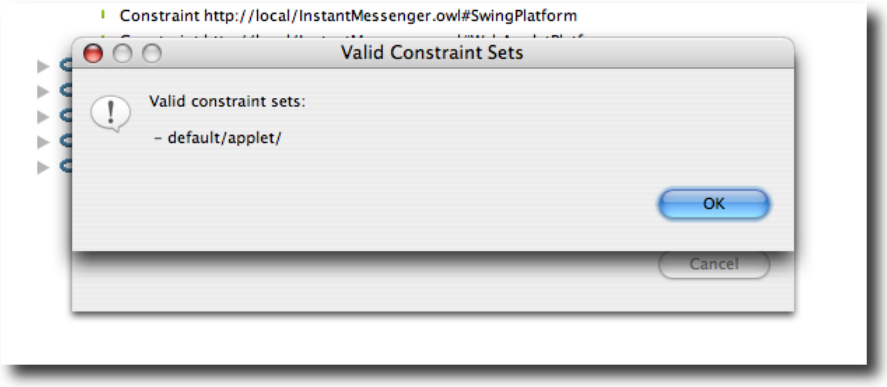

Figure 7.35: Validate result

multiple modules. These modules can then be loaded on top of each other, after which they are executed as one transformation module. Superimposition has been implemented as an ATL byte code transformation that is applied at load time. Normally, every transformation module has a `main()` operation. With superimposition, multiple transformation modules are loaded in sequence, after which the initial `main()` operation is adapted such that the newly loaded rules are invoked and additional helper attributes are initialised.

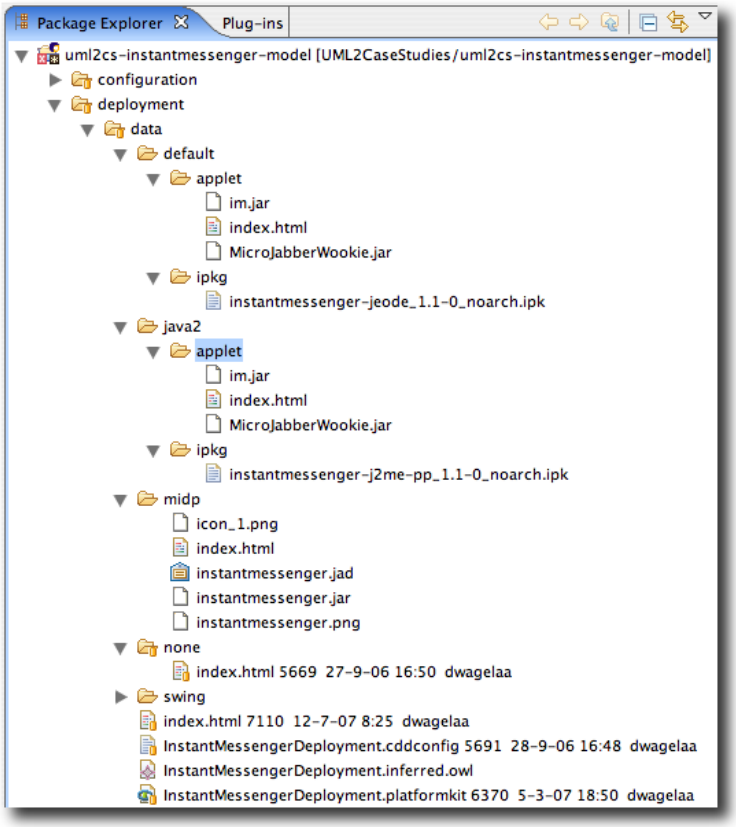

Figure 7.36: Deployment folder for the instant messenger product line

The `main()` operation of each superimposed module is then discarded. The result is one `main()` operation that ensures all rules are invoked (and helpers initialised) in their load time order (i.e. superimposition order).

Superimposition is implemented in a single Java class[18]. An instance of that class is used in the ATL launcher code. Superimposition uses the internal object representation of the ATL byte code (as used by the ATL virtual machine) for its transformations.

7.4.2 Modularised meta-models

ATL is based on some simple assumptions of what models and meta-models are. Two of these assumptions are:

1. A model is contained in a single file and has a single meta-model.

2. A meta-model is a model of the models that conform to it.

[18] https://bugs.eclipse.org/bugs/show_bug.cgi?id=156095

As a consequence of these assumptions, meta-models are also assumed to be contained in single files. This is often not the case for EMF-based meta-models, such as the Eclipse UML2 meta-model. In EMF, model elements are allowed to reference elements inside other models, which reside in other files. The Eclipse UML2 meta-model references the Ecore meta-model and defines its meta-classes on top of it.

ATL bases itself on the single file meta-model assumption to globally access all meta-classes inside a meta-model. If a meta-class cannot be directly retrieved from the meta-model file, it does not exist for ATL. In Eclipse UML2 models, instances of other meta-classes than those in the specified meta-model file can occur, such as Ecore EAnnotations or specific stereotype applications (see also subsection 7.4.3). These instances cannot be directly retrieved – or created – in an ATL transformation module.

We have tackled this mismatch between ATL and EMF[19] by doing an exhaustive and deep traversal over all meta-classes of the specified meta-model file. Since EMF automatically resolves and loads additional models as soon as they are accessed, we only need to record which files have been loaded by EMF. From now on, ATL retrieves a meta-class either from the specified meta-model file or from any of the referenced meta-model files. For ATL/EMF, a meta-model is now effectively defined by the transitive closure of references made from the meta-classes in the initially specified meta-model file. The resulting meta-model is contained in a set of files that contain all the initial meta-classes as well as all referenced meta-classes.

7.4.3 Stereotypes as meta-classes

As has been explained in chapter 2, Eclipse UML2 stereotypes have a dual representation. Each stereotype exists as a UML2 Stereotype instance as well as an Ecore EClass instance. In EMF, all meta-classes are instances of EClass. That means that each stereotype has a meta-class representation. This meta-class representation is used to create stereotype *applications*, which are effectively instances of the meta-class.

ATL does not support stereotype applications directly. Instead, ATL must use the Java API of Eclipse UML2 to apply stereotypes. This is because stereotype applications are "strange" model elements for ATL, of which the meta-class is not part of the meta-model. The meta-class representation of stereotypes are actually embedded in the UML profiles.

We have adapted ATL[20] to be able to read an Eclipse UML2 profile as an EMF meta-model and retrieve the contained meta-classes. In combination with modularised meta-models, as described in subsection 7.4.2, it is now possible

[19] https://bugs.eclipse.org/bugs/show_bug.cgi?id=140546
[20] https://bugs.eclipse.org/bugs/show_bug.cgi?id=156093

to define a "placeholder" meta-model that references the Eclipse UML2 meta-model as well as all applicable UML profiles. In this way, ATL has access to all UML meta-classes as well as all stereotype meta-classes.

7.4.4 Debugging support for multiple transformation modules/libraries

The Eclipse visual debugger for ATL only has support for a single ATL transformation module as far as source code lookup is concerned. This means that one cannot step through library code when doing source-level debugging. This is mainly a granularity problem, where only top-level code from the transformation module was accessible through the debugger.

Ever since ATL supports module superimposition, however, the problem is more than just a granularity problem. All source code that resides in superimposed transformation modules is not accessible through the debugger. We have adapted the ATL debugger[21] to support explicit lookup of the actual ATL source code file that contains the code that is currently stepped through. As a result, the ATL debugger supports stepping through code of superimposed modules as well as libraries.

7.5 Limitations and future work

Even though several tools have been discussed in this chapter, the PlatformKit tool is central and implements the ideas presented in this dissertation. We will therefore focus on the PlatformKit tool. This section discusses its limitations and suggests how these limitations can be mitigated. Directions for future work will also be discussed.

7.5.1 Performance and memory usage

PlatformKit uses the Jena framework for loading, saving and manipulating ontologies. While it is a very powerful framework with support for RDF, RDFS, OWL, local ontology caching/aliasing and ontology reasoning, it is not the fastest and most efficient framework available. Whereas DL reasoning is the most complex task in theory, looking up OWL classes and saving OWL ontologies in Jena is the most time-consuming task in practise. Memory usage currently lies around 500 MB of RAM. The OWLAPI[22] framework promises more efficiency at the cost of some features. At the time of this writing, OWLAPI does not support the DIG reasoner interface.

[21] https://bugs.eclipse.org/bugs/show_bug.cgi?id=192445
[22] http://owlapi.sourceforge.net/

PlatformKit currently uses a package-level OWL representation of the various Java APIs. Experiments with class-level OWL representations have shown that the current tool setup (Jena/Pellet) doesn't scale up to this level of granularity. No results have been obtained within a time limit of 2 hours and a memory limit of 2 GB RAM on current Intel Core2 Duo hardware @ 2 GHz (our design-time setup for creating ontologies). ATL model transformations have been used to translate a finest-level granularity representation (attributes, operations, parameters, etc.) of the Java APIs in UML to OWL. These model transformations create subsumption and equivalence relationships in the process, indicating that they can be used as an alternative to OWL class hierarchy inference. The fact that ATL can still deal with the finest-level granularity and Jena/Pellet already give up at class-level granularity, demonstrates that ATL model transformations scale better. That said, OWL is more declarative than an ATL model transformation. We also have not tested with a commercial DL reasoner, such as RacerPro[23].

7.5.2 Automatic platform discovery

PlatformKit currently requires a user to provide an explicit platform description ontology. This is not a realistic requirement for the platform-driven deployment scenario in which an appropriate software configuration is automatically chosen and deployed. While PlatformKit already recognises a number of standard platforms, for which it can retrieve a platform description from its database, many platforms are too dynamic to capture in this way. A regular desktop PC with a certain web browser, for example, doesn't say anything about whether or not Java is installed and if so, which version of Java.

Most dynamic web sites solve this problem by including a script that can detect the presence of required web browser plug-ins. Depending on what kind of software we want to automatically deploy, we can provide platform discovery software that can either be run directly from the browser or downloaded and run on the target platform. While such discovery software has its own platform dependencies, the range of platforms we need to support typically leans towards the higher end. Constrained platforms, such as mobile phones, offer a very limited platform, but that platform is also much more static. A static platform means that it is guaranteed to contain certain features, such as a specific Java version and a specific web browser version. For such constrained platforms, we can fall back on the existing PlatformKit functionality that recognises standard platforms by their web browser identification. For the scope of high-end Java platforms, there is enough common functionality to implement a single platform discovery agent that runs on all high-end Java versions.

[23] http://www.racer-systems.com/

7.6 Summary

In this chapter, we have described the architecture of PlatformKit and how it can be used to perform various tasks. PlatformKit consists of two main components: an Eclipse plug-in that is used during design-time and a web servlet that is used during deployment. The tasks cover the different steps in setting up a model-driven software product line, such that platform dependency constraints are taken into account. The final step then allows to automatically deploy a specific software product configuration based on its platform dependency constraints.

As a limitation, high CPU and memory usage have been discussed. The reason for this limitation turned out to be not so much the DL reasoner, but rather the ontology manipulation framework Jena. To keep CPU and memory usage within acceptable limits, the platform vocabulary ontologies are kept at a certain level of granularity. The Java API ontologies are at package-level granularity, for example. A direction for future work is automatic platform discovery, where a client-side platform discovery agent is used to provide a platform description ontology for the more dynamic high-end platforms.

Chapter 8

Conclusion

8.1 Summary

Today's software systems are more and more often required to run on multiple platforms. This puts an extra burden on the software developer, who now has to build and maintain a separate version of the software for each target platform. Even when looking at a technology like Java, which was meant to overcome platform differences, we find that multiple versions of the Java platform exist. The family of Java platforms ranges from J2ME for mobile devices to J2EE for server-side enterprise applications.

Tomorrow's software systems are described in the vision of *Ambient Intelligence*, which aims for a user-driven, service-based computing environment that includes personal devices as well as special-purpose embedded devices in the environment. This vision only amplifies the current platform diversity, as the hardware and software combinations in personal and special-purpose devices vary greatly.

The Model Driven Architecture (MDA) proposes a solution to this problem. Software is modelled in high-level Platform-Independent Models (PIMs). Model transformations are used to refine the PIM into a Platform-Specific Model (PSM). PIM-to-PSM transformations can be performed in multiple refinement steps. Each of these PIM-to-PSM refinement transformations introduces certain platform dependencies into an intermediate PSM. The final PSM often takes the form of generated code.

The MDA suggests the use of a Platform Model (PM) to drive the PIM-to-PSM transformation. In practise, no PM is used: the platform dependencies introduced by PIM-to-PSM transformations remain implicit. As a result, it is not safe to reuse a PIM-to-PSM transformation for other platforms than the one for which it was written. The only safe assumption is that each targeted platform requires its own dedicated PIM-to-PSM transformation. This causes a large maintenance burden for each supported platform.

A PIM-to-PSM transformation can be split up in multiple, step-wise refinement transformations. These transformations can be organised in a workflow or build script that executes each refinement transformation. Individual refinement transformations can be reused for other platforms. While these separate refinement transformations lay the basis for lifting the platform maintenance burden, they do not solve our initial problem. It is still unclear whether a refinement transformation can be used for other platforms than the one for which it was written, since all platform dependencies are still implicit.

As the software developer composes the build script that executes the refinement transformations, the platform dependencies that each refinement transformation introduces must be considered as well. One approach is to test the generated software on the target platform to tell if the software works on that platform. Testing on each platform is a time-consuming activity, however, and may even leave certain incompatibilities undetected until after deployment. Another approach is to use an automated configuration process that enforces the satisfaction of platform dependency constraints. Such a configuration process does not exist for the MDA today.

This dissertation has proposed to solve these problems by using an explicit platform model that is used in an automated configuration process. This platform model captures platform domain knowledge that allows a software developer to express platform dependency constraints as well as platform instances. It is expressed in the Web Ontology Language (OWL), which is an extensible language for describing ontologies. Ontologies are commonly used to represent domain knowledge and to provide a community of users with a controlled vocabulary. We use the OWL DL variant, which allows us to apply automatic reasoning.

An automated configuration process based on Software Product Lines (SPLs) is proposed. SPLs integrate a number of software-intensive products that share a significant amount of functionality. As such, any software that is developed using the MDA approach can be considered as an SPL, since each platform-specific software product shares significant functionality with other platform-specific versions of that software product. In contemporary SPL practise, Product Models are used to automatically generate the software products. These Product Models can be expressed using a Domain-Specific Modelling Language (DSML). In the MDA, the Product Model translates to a Configuration Model that states which PIM-to-PSM refinement transformations must be applied. Instead of generating a software product, the build script that executes the refinement transformations is generated. The generator itself is implemented as a model transformation, which allows for using the same model transformation technology as was used for the PIM-to-PSM refinement transformations.

8.2 Thesis statement

The discussed propositions follow the thesis statement of this dissertation:

> *In order to deal with platform diversity, we believe that* ***(1) platform domain knowledge must be made explicit****. By making this knowledge explicit, we can* ***(2) reason about the extent of platform dependencies with regard to platform instances*** *as well as* ***(3) compare appropriateness of alternative refinement transformations*** *and* ***(4) enforce a safe configuration of refinement transformations via their introduced platform dependencies****.*

Our solution to the discussed problems is centred around an explicit platform model (1). Our choice of OWL DL as a representation language for this platform model provides us with reasoning capabilities. We have chosen a specific OWL representation for platform dependencies and platform instances, such that a DL reasoner can tell us whether a platform dependency is satisfied by a platform instance (2). In addition, our representation of platform dependencies allows us a DL reasoner to organise them in a hierarchy that represents appropriateness with regard to a targeted platform (3). Our integration of the platform model with an automated configuration process enables us to enforce that a configuration has all its platform dependencies satisfied (4).

The following section discusses each of the contributions of this dissertation in detail.

8.3 Contributions

8.3.1 A common platform domain model

> We have presented how OWL-DL can be used to define a general platform model that serves as a common ontology for specific platform sub-domains [PVW+04], [Wag05], [WJ05].

In current MDA practise, the platform model that is meant to drive PIM-to-PSM transformations remains implicit. We proposed to make that platform model explicit, such that we can reason about platforms and platform dependency constraints. We chose the Web Ontology Language (OWL) to represent our platform model. We use the OWL DL variant, which corresponds

to description logic (DL) and allows us to apply automatic reasoning. Our usage of OWL DL is currently limited to $\mathcal{SHF}(\mathbf{D})$ instead of the standard $\mathcal{SHOIN}(\mathbf{D})$. This allows for a slightly better reasoning performance than is possible with standard OWL DL, depending on the reasoner implementation.

> We have also presented how the common platform model can be extended for the platform sub-domain of Java Runtime Environments (JREs) [Wag05], [WJ05], [WV07].

We have sub-divided our platform model into a hierarchy of ontologies, with a general *platform* vocabulary at the root. The general platform ontology provides the concept of "Platform", its various properties and related concepts. On top of this ontology, we have built a top-level ontology for Java platforms. This Java ontology introduces the concept of a Java Runtime Environment and related concepts and properties. In order to reason about specific versions of Java, we have introduced an extra ontology for each version of Java we want to reason about, such as a J2SE 1.3 ontology and a J2ME MIDP 1.0 ontology. The ontologies for the specific versions of Java have been automatically generated from the class libraries that are enclosed with each Java version.

8.3.2 A method for describing platform dependencies and platform instances

> We have presented how platform dependency constraints and platform instances can be described, based on the common platform model and extensions. The platform dependencies can be compared against platform instances to check if they are satisfied. In addition, we can determine which platform dependencies are *more specific* than other platform dependencies and form a closer match to the targeted platform [Wag05], [WJ05], [WV07].

The explicit platform model we have proposed allows us to describe explicit platform dependency constraints and platform instances. The platform dependency constraints are represented as OWL classes. Those OWL classes can be automatically organised into a subsumption hierarchy, where the root classes in the hierarchy are *least-specific* to a platform and the leaf classes are *most-specific*. This allows us to optimise with regard to platform specificness when a choice has to be made from a number of alternatives. Platform instances are represented as OWL individuals. A platform dependency constraint is considered satisfied if there is at least one OWL individual that can be classified as an instance of it.

8.3.3 A framework for platform dependency management

We have presented how the platform model can be integrated in a software development process based on the MDA and SPLs. The relationship between the MDA and SPLs has been explored as part of this work and a common configuration approach based on DSMLs has been used [WV06], [WV07].

Explicit platform dependency constraints are only useful if they can be enforced during the configuration of PIM-to-PSM refinement transformations. As the MDA offers no process for the configuration of step-wise PIM-to-PSM refinement transformations, we have introduced a configuration process with automated support for platform dependency constraints. This has resulted in a *framework for managing platform dependencies* for the MDA that:

- improves the maintainability of a PIM-to-PSM transformation configuration by enabling safe reuse of individual refinement transformations in such a configuration,

- integrates with existing software development technologies, in particular the software development technologies that target Java platforms.

The first property is achieved by annotating the meta-classes in a configuration DSML meta-model with the appropriate platform dependency constraints. Each (concrete) meta-class refers to a configuration choice. A meta-class that is annotated with a platform dependency constraint refers to a configuration choice that introduces platform dependencies. This approach allows us to "profile" the configuration language meta-model against a platform instance, such that only the meta-classes that are valid for that platform can be instantiated. The result is a PIM-to-PSM transformation configuration that is validated to work for a given target platform. In addition, we can check pre-existing configuration models against a platform instance and check if all instantiated meta-classes are valid for that platform. The result is a set of configuration models that is validated for a given target platform. This scenario is particularly useful when one has to choose from a number of alternative, pre-generated configurations during the deployment phase.

8.3.4 A framework for platform-driven optimisation

We have presented how platform dependencies can be used as the basis for selecting optimal model transformations (or SPL features). In addition, we have presented how this can be extrapolated for the selection of optimal configurations [Wag05], [WJ05], [WV07].

Our approach for describing platform dependency constraints in OWL has laid the basis for platform-driven optimisation. We can determine the class hierarchy for a number of platform dependency constraints and derive which platform dependency constraints are *most-specific* and *least-specific* with regard to the platform. We have integrated this optimisation strategy in our configuration framework, such that it:

- assists in finding the most appropriate PIM-to-PSM transformation for a specific platform from a number of alternatives.

We have introduced an intermediate "shadow" model for this purpose, called a PlatformKit model. The PlatformKit model allows us to semi-automatically sort a list of platform dependency constraints *most-specific-first* or *least-specific-first*. It can also do this for *groups* of platform dependency constraints. This is useful for comparing entire configuration models against each other, where each configuration model can have multiple platform dependency constraints. The sorting process is semi-automatic, since it is not always possible to determine a *more-specific* / *less-specific* relationship between two platform dependency constraints. The sorting process therefore uses a manually pre-sorted list as its input and makes a minimal amount of changes to that list.

This optimisation strategy can be used in the two scenarios described in the previous subsection: when "profiling" a configuration language against a platform instance, the configuration language editor can be adapted such that the list of configuration choices is sorted *most-specific-first*. During the deployment scenario, one has to choose from a number of alternative, pre-generated configurations. The resulting set of validated configuration models can now be sorted *most-specific-first* for the purpose of deploying the configuration that lies closest to the target platform.

8.3.5 A case study that applies the explicit platform model in an MDA/SPL setting

We have developed a non-trivial case study of a cross-platform Instant Messaging client that demonstrates the merits and limitations of our approach [Wag05], [WJ05], [WV07].

Throughout this dissertation, we have used the example of an Instant Messaging client that runs on various Java platforms. This example has been worked out in detail as a case study and can be found at `http://ssel.vub.ac.be/ssel/research:mdd:casestudies`. The case study is situated within an Ambient Intelligence context, where people with heterogeneous personal devices can download and use the Instant Messaging client. The case study applies several alternative PIM-to-PSM transformations, each of which are tailored towards a specific aspect of the Java platform family. The case study also employs several optional and/or alternative SPL-style features in the form of additional models that build on top of the main PIM. In our case study, these models are *semi*-platform-dependent, as the modelled feature is only relevant to a subset of all targeted platforms. Finally, a configuration language has been developed for the Instant Messenger SPL. This configuration language demonstrates the use of platform dependency constraints during the configuration and deployment of the Instant Messaging client.

8.3.6 Tool support

We have developed a tool, named *PlatformKit*, that implements platform dependency management and platform-driven optimisation based on the Eclipse Modeling Framework [WV07]. In addition, we have developed the *Jar2UML* tool that reverse engineers Java class libaries to UML models. Finally, we have added several improvements to the ATLAS Transformation Language tool for the purpose of our case study [Wag08].

All our experiments have been carried out on the Eclipse platform. The Eclipse IDE already provides a number of plug-ins for modelling (EMF), model transformation (ATL) and Java development (JDT). As Eclipse is built on top of Java, we were able to use additional Java frameworks for the manipulation of OWL ontologies (Jena) and automatic DL reasoning (Pellet). We have built on top of these technologies for the implementation of our PlatformKit Eclipse plug-in. For the PlatformKit deployment servlet, we have used MySQL and

Jetty in addition. PlatformKit implements the configuration and deployment scenarios described earlier. The platform instances that are required for these scenarios can be obtained in several ways:

- built-in platform descriptions,
- manually provided platform descriptions,
- database of known platform descriptions.

The first two of these can be used with the PlatformKit Eclipse plug-in during configuration. During deployment with the PlatformKit servlet, only the last two options are available. The database of known platforms uses web browser agent IDs to look up the corresponding platform description. This works particularly well for small portable devices, where the software platform is often fixed (e.g. mobile phones).

The Jar2UML tool is used to reverse engineer tailored UML models from Java class libraries, such that platform ontologies can be generated from them. In addition to UML models of Java class libraries, Jar2UML can also extract all dependencies from a jar file as a UML model. PlatformKit can use these UML models to determine compatibility of the jar file with a particular Java class library. Jar2UML is part of the Eclipse MoDisco reverse engineering tool box[1].

Several improvements have been made to the ATL tool for the purpose of our instant messenger case study. These improvements have also proven useful beyond the case study, as they are integrated back into the main ATL source code. Because of the impact and number of improvements, we have received committer access to the ATL source code such that future improvements can be propagated more efficiently. One of the improvements, module superimposition, is described in detail in [Wag08].

8.4 Reflection

8.4.1 Generative vs. reflective adaptation

In chapter 1 we have introduced two approaches for adapting software to a target platform. The generative approach transforms the software beforehand to fit a target platform. The reflective approach reacts to platform differences at run-time. Each of these approaches have strengths and weaknesses. Generative approaches typically can't deal with post-deployment changes in the platform. Reflective approaches inevitably have some overhead in doing run-time platform checks and providing code that may never be executed on

[1] http://www.eclipse.org/gmt/modisco/toolBox/

the target platform. We have chosen to follow the generative approach in the form of the MDA in this dissertation. This choice has given us more room to work with heavy-weight reasoning and configuration approaches that are not always feasible at run-time. At run-time, the platform is intended for the actual applications and any platform dependency reasoning and configuration is overhead.

That does not rule out the use of our approach for reflective adaptation, however. In reflective adaptation, the platform dependencies are also known beforehand, as they are explicitly checked for in the code. Our platform vocabulary ontologies can provide a language-independent separation between discovering the platform through reflection and matching that to a platform dependency. In addition, our platform dependency reasoning mechanism can help determine the optimal adaptation to the target platform through determining the *most-specific* platform dependency. This can be done beforehand, since the platform instance information is not necessary for this step.

To see if our configuration approach can also be used for reflective adaptation, it is best to look at an example SPL that does reflective adaptation. The Eclipse platform is essentially an SPL that allows for configuration and adaptation at run-time. This is done via the service discovery functionality provided by Eclipse's OSGi implementation[2]. The configuration language of the Eclipse platform is defined by *extension points*[3]. Extension points provide a typed configuration interface that will take only specific extensions. Extensions to the platform are typically provided by Eclipse features that contain plug-ins. Plug-ins can in turn define new extension points. When installing a new Eclipse feature, the Eclipse platform checks if the dependencies of that feature are satisfied by the running Eclipse configuration. Often, multiple versions of a feature are available, such as a "run-time" version and an "SDK" version. The user currently has to decide which version is best for his/her situation. Eclipse only checks if the chosen version fits in the current Eclipse configuration.

Our platform models can help to provide an Eclipse user with feedback on which is the *most-specific* version of an Eclipse feature that still works for his/her Eclipse configuration. This automated assistance in turn also opens the door to providing more variants of an Eclipse feature, because users will be supported in their selection process anyway. As an example, we'll consider the Hibernate Tools for Eclipse and Ant[4]. These tools are available as an Eclipse feature and include a wide array of tools, ranging from Ant tasks for Java code formatting to integration with Eclipse's Web Tools Platform (WTP). Unfortunately, this wide array also brings with it a host of dependen-

[2] http://www.eclipse.org/equinox/
[3] http://www.eclipse.org/resources/?category=Extension%20points
[4] http://tools.hibernate.org/

cies: Eclipse Ant Tools, Java Development Tools, Web Tools Platform, etc. A user that only requires the Ant tasks for Java code formatting is still forced to satisfy all the dependencies of the Hibernate Tools for Eclipse. With our platform models, it becomes feasible to create multiple variants of the Hibernate Tools to better match the requirements of different users. The platform dependency constraints for each variant can be used to pre-sort a list of variants in *most-specific-first* order. All variants with unsatisfied platform dependency constraints can then be marked as unsatisfied[5]. The user can now decide to choose the variant that fits his/her current Eclipse configuration best. Alternatively, the user can choose to find a variant that lies close to the optimal variant, but has less platform dependency constraints or adds some extra (unsatisfied) platform dependency constraints.

Deployment of an Eclipse feature on a running Eclipse platform is an example that stays close to our current configuration approach. Another Eclipse example illustrates more typical reflective adaptation: the Subclipse[6] feature for Subversion repository access from Eclipse includes two ways of accessing a Subversion repository. One way is through a pre-installed, native binding to the "`svn`" program ("JavaHL"). Another way is by using "SVNKit", which is a Subversion library that is completely written in Java. Our platform model can be used here to determine the *most-specific-first* order in which to try the alternatives. Checking platform dependency satisfaction is not useful here, since we can simple "`try`" one approach and "`catch`" up when it turns out not to work: exception handling allows us to adapt our behaviour and switch to our next option. In cases where the platform dependency constraints refer to required CPU power and memory, such a trial-and-error approach cannot be used. Recovery from resource starvation, if possible, can take very long and affects other running software. Explicit dependency satisfaction checking is useful here, especially when a platform vocabulary ontology is available that translates, e.g. specific CPU models to required performance figures.

8.4.2 Platform modelling language

We have chosen to use OWL DL to model platforms, platform dependencies and platform instances. The part of our platform models that provide a general description of the platform domain is an ontology by its very nature, and we have called that part the "platform vocabulary ontology." It is therefore natural to express that part in an ontology language. Platform dependency constraints and platform instances are most easily expressed in the same language as the platform vocabulary ontology on which they are based. Other formats are available to express our platform knowledge, however. Examples are RDF,

[5]Eclipse offers functionality to find and include necessary dependencies, so unsatisfied variants must still be available for selection.

[6]`http://subclipse.tigris.org/`

Prolog and Constraint Satisfaction Problems (CSPs), as already discussed in the related work section of chapter 6. OWL is actually based on the RDF language, where RDF is limited to individuals with property values, called "triples". An example RDF triple is ("Eric" "hasMailBox" "em@w3.org"). One can apply simple, value-based reasoning to RDF statements. Prolog offers a light-weight method for expressing platform dependency constraints and platform instances (provided that we don't build an entire DL reasoner on top of Prolog). CSPs are more heavy-weight and intended for solving NP-hard combinatoric problems.

None of these approaches provide us with both guaranteed complete reasoning results and a classification of platform dependency constraints in a *most-specific/least-specific* hierarchy. In contrast to *feature interaction constraints*, we don't need to solve combinatoric constraint satisfaction problems: an area in which CSPs are the tool of choice. Instead, our platform dependency constraints typically point to external facts that are unaffected by our configuration choices. In chapter 4, we have shown that the DL constructs we require justify using OWL DL as a language. We currently require the $\mathcal{SHF}(D)$ DL, which is most closely matched by the standard OWL DL language ($\mathcal{SHOIN}(D)$). We have also shown that we can reduce our usage of DL constructs to $\mathcal{ALCF}(D)$, if we choose to deviate from standard DLs and the reasoning systems that implement them. This allows us to classify our platform dependency constraints and determine their satisfaction in PSPACE complexity [Tob01], and shows that any scalability problems with our approach can be solved. However, the platform ontologies used in our instant messenger case study have already proven to scale sufficiently to be usable.

The extra expressiveness of the $\mathcal{SHF}(D)$ DL we currently use provides us with some benefits. We can use property hierarchies to evolve new properties from existing ones without changing property instances, for example. Transitive properties are in turn useful to determine the transitive closure of required features for any given feature. Coupled with the availability of highly optimised reasoners for the standard $\mathcal{SHOIN}(D)$ DL, this provides a good compromise between expressiveness and performance.

8.4.3 Scope and reusability of our approach

Our approach is composed of several parts with varying scopes and reusability. The reasoning mechanism discussed in chapter 4 can be applied to any platform, as long as we have a platform ontology for it. The configuration mechanism discussed in chapter 6 can be applied to any SPL, either or not based on the MDA, as long as we use a configuration modelling language. With minimal change, we can also support configuration that is not based on a configuration modelling language, such as feature models or textual configuration languages. It is the platform vocabulary ontology that limits the scope and

reusability of our approach.

While we have started from a very general platform ontology that describes concepts such as "Platform", "Feature", "Software" and "Hardware", we quickly went into detail after that. The other platform vocabulary ontologies we have created refer to specific Java runtime environments. We have also illustrated through our examples that the most specific platform vocabulary ontologies provide the most expressiveness, such as the "jdk-1_1:JavaUtilLibrary" concept that represents the libraries that implement the JDK 1.1 version of the `java.util` package specification.

If we want to express platform instances and platform dependency constraints for another domain than the Java runtime environments that we have covered, we must provide additional platform vocabulary ontologies. In section 8.5, we name a few possible candidates for additional platform vocabulary ontologies. In this light, our reasoning and configuration mechanisms and accompanying tool support can be considered as a commodity. They are therefore available as open source software. The real value lies in the in-depth platform domain knowledge that is encoded in the platform vocabulary ontologies. Our contribution is to show how platform vocabulary ontologies can be constructed. The specific knowledge of particular platform domains serves only particular market segments, such as the Java embedded market segment, the Eclipse plug-in market segment or the Enterprise Java market segment. The scope of a platform vocabulary ontology should be derived from the scope a such a market segment: only the platforms that are covered by the targeted market segment must be modelled.

Another aspect of platform vocabulary ontology scope is the level of detail we have modelled within a single platform domain. In our platform ontologies of Java runtime environments, we have modelled the API up to the package level, but not below (classes, methods, etc.). This is a trade-off between the detail of dependencies and the performance of dependency satisfaction checking. We follow the same approach as, for example, OSGi uses for expressing dependencies of so-called Java *bundles*[7].

8.5 Future work

8.5.1 Automatic platform discovery

The way in which platform descriptions are currently provided to PlatformKit is not unobtrusive and can even be daunting. Especially during deployment, it is important that a platform description can be easily provided, preferably without the user knowing. The current methods for providing platform descriptions are limited to:

[7] http://www.eclipsezone.com/eclipse/forums/t90544.html

Built-in platform descriptions: The PlatformKit tool comes packaged with a number of pre-defined "prototype" platform descriptions. These include standard PCs, PDAs and mobile phones with standard Java environments. These prototype platform descriptions only provide a minimal amount of data, such that the description fits a lot of "real" platforms. Built-in prototype platform descriptions are typically used during development to make sure that the developed software stays within specific platform boundaries.

Manually provided platform descriptions: It is possible to hand-craft a platform description in OWL and submit it to PlatformKit. This possibility provides a maximum amount of freedom, since one can submit any OWL document as a platform description. This method requires in-depth knowledge of our platform vocabulary and how it can be used. Normal developers are not expected to use this method, but this method is interesting for software architects that want precise control over the exact range of platforms they want to support.

Known standard platform descriptions: PlatformKit supports a database of platform descriptions. These platform descriptions are indexed by the "user-agent" identifier string of the platform's web browser. This method is especially interesting for "fixed" platforms where one particular web browser version uniquely identifies the entire platform. This is typically not the case for standard desktop computers, but it is almost always the case for mobile phones. In fact, almost any device that operates from a "firmware" (i.e. a monolithic software platform) can follow this approach.

Only the known standard platforms can cover a wide range of platforms, whereas the built-in platform descriptions and manually provided platform descriptions cover a narrow range only. In addition, most platforms consist of a dynamic, user-selected mix of hardware and software components. The known standard platforms approach cannot be applied for those platforms.

It would therefore be useful to automatically discover the particulars of a target platform without user intervention. A small, cross-platform bootstrap program could be used, for example, to perform this platform discovery task. In the domain of Java platforms, Java applets can run on any Java client platform except J2ME MIDP (mobile phones). That sufficiently covers the range of platforms with a dynamic mix of hardware and software.

8.5.2 Setting up additional platform ontologies

Our platform ontologies currently only cover a small portion of all relevant platforms. Detailed ontologies only exist for Java platforms – a subset of all

Java platforms at that. The main Java client platforms, such as J2SE, J2ME and their variants, are modelled in detail. There exist several extensions to these Java platforms that are interesting in an Ambient Intelligence environment, such as the Java Media Framework, the J2ME Messaging API, Remote Method Invocation for J2ME, Java3D and others. Often, there exist vendor-specific versions of these Java extensions as well, such as the Siemens 3D API for their J2ME implementation. Most mobile phones have some API or another for 3D graphics under Java via such an extension. Developers tend to stay away from such extensions, however, because of having to support all the different alternatives and the maintenance problems that come with it. Platform ontologies of the different alternative APIs can help alleviate these problems in the same way that the main Java client platform ontologies alleviate platform dependency maintenance problems. Detailed Java platform ontologies can pinpoint the exact class of platforms for which a particular Java extension is valid. They allow for a finer-grained platform support than what the current Java platform ontologies can achieve.

Whereas the case of Java platform extensions illustrates the need for in-depth elaboration of the platform ontologies, there is also a need to widen the scope of our platform ontologies. Enterprise Java platforms are typical targets for the MDA and also often have vendor-specific extensions. The reason that the MDA is so popular in the enterprise domain is that the software under development must typically survive a number of platform evolution steps. That brings an extra dimension of platform diversity. Finally, there are other relevant platform technologies than the family of Java platforms. Microsoft's .NET platform family offers similar functionality to Java, and there are plenty of other, often smaller, platforms that offer cross-platform functionality (Squeak/SmallTalk, Python, Ruby, etc.). There are even certain C/C++ libraries that are worth modelling in a platform ontology. Those libraries typically try to be cross-platform – where possible. An example of such a library is the Simple Direct Media (SDL) layer, which attempts to provide a cross-platform interface to all sorts of multimedia hardware (graphics, sound, ...). Modelling the subtle differences in the use of SDL on different platforms in a platform ontology alleviates some of the maintenance problems of developing a cross-platform computer game, for example.

8.5.3 In-depth analysis of ATL improvements

Whereas we have made several improvements to ATL, we have only done our first analysis of module superimposition in [Wag08]. Our improvements to ATL, as described in chapter 7, and uses of ATL as applied to our instant messenger case study and the PlatformKit tool deserve more in-depth analysis and comparison to other approaches.

Whereas ATL module superimposition has already been described and com-

pared to other composition approaches, only two application areas of module superimposition have been charted. We use module superimposition mainly to define refinement transformations (e.g. UML2Profiles) on top of a copying transformation (e.g. UML2Copy) in our instant messenger case study. In addition, we use module superimposition to refine a general transformation into a specific one by overriding and adding rules (e.g. UML2ToAPIOntology and UML2ToPackageAPIOntology in appendix A). Another use of superimposition we want to look into is leverage tracing information [CH06]. ATL is one of the transformation languages that keep internal tracing information, which means that ATL keeps an automatic record of which source model elements are transformed to which target model elements. This tracing information is only available during the execution of the model transformation and cannot be accessed later by other model transformations. Superimposition provides a way to run multiple model transformations together, where all tracing information is available to all model transformations being executed.

The capability of ATL to use EMF meta-models spread over multiple files allows for a powerful abstraction to be made. Meta-models are no longer tied down to files, but instead just have a starting "address" (URI) from where ATL starts collecting the meta-model. Such an abstraction may be useful to other transformation approaches, such as graph transformation [MG06]. Graph transformation is an in-place transformation approach that operates on a single model. If we can apply the multi-file abstraction to that single model notion, we may enable graph transformations to access multiple models (in multiple files) as if they were a single model.

The capability of ATL to access meta-class representations of stereotypes allows direct manipulation of stereotype applications. In the current situation, however, UML profiles must be explicitly mentioned by a "placeholder" meta-model, which is inconvenient and difficult to understand for most users. Eclipse UML2 profiles also have a certain versioning system that allows the existence of multiple versions of the same meta-classes. This versioning system is meant to mitigate the transition problem of models that refer to (the old version of) the UML profile. The concept of multiple meta-class versions is not understood by ATL, however. We have to analyse what it means to have complete support for UML profiles. Any such complete UML profile support in ATL should also be done in a general way, without ATL having any UML-specific knowledge. As a contrast, the current complete UML profile support for ATL is implemented as a specific modelling framework driver[8]. This is an obvious mismatch, as UML is not a modelling framework itself, but is instead built on top of the EMF modelling framework.

Finally, we have used a *higher-order* transformation (EModelCopyGenera-

[8] http://wiki.eclipse.org/ATL_Model_Handlers

tor[9]) to generate a copying transformation (UML2Copy) for UML models from the UML meta-model. This approach was taken such that our case study could deal with PIM-to-PSM refinement transformations where more than one model is involved (e.g. UML libraries and profiles). ATL already offers a special "refining mode" for situations that involve only one source model. Higher-order transformations can often be used to achieve the same effect as a built-in ATL features, such as refining mode or module superimposition. They deserve more in-depth investigation as to what effects can be achieved and how they can be used to provide an executable and well-defined specification of new ATL language features.

8.5.4 Application in other domains

We have applied our platform ontologies to a fairly traditional software development domain: that of Java client applications. The focus of the platform ontologies themselves was mostly on API, where the remaining ontology information served mainly to situate APIs within the domain of platforms. There are other interesting application domains for platform ontologies than APIs. One example of such a domain is web service orchestration, where a workflow model is used to tell an orchestration engine which web services must be invoked and how their results must be combined.

Web-services orchestration engines may support different workflow features. A platform ontology of different orchestration platforms can support the use of advanced and optimised orchestration workflows that make full use of the targeted orchestration engine. Platform dependency constraints can enforce the validity of orchestration workflows for a target orchestration engine. Since the workflows are typically expressed as models, most of our framework can be applied directly to this domain. Standard workflow modelling languages, such as BPEL, can be extended for a specific orchestration platform. Model transformation can be used to refine platform independent workflows to platform-specific workflows.

In order to leverage platform-specific workflow features, the domain of orchestration engines must be researched and modelled in a platform ontology. First, a general vocabulary ontology of orchestration engines must be developed. This ontology must be extended for a number of real orchestration engines, such as ActiveBPEL, Twister, JBoss jBPM, bexee or MOBE. The most leverage is typically achieved when modelling orchestration engines with a significant overlap in functionality, such as engines that all claim to support BPEL, but only up to a certain extent. Based on these platform ontology extensions, we can define a number of prototype orchestration platform descrip-

[9] http://ssel.vub.ac.be/viewvc/UML2CaseStudies/uml2cs-transformations/EModelCopyGenerator.atl?view=markup

tions, such that we can develop for a specific class of orchestration platforms.

Appendix A

Ontology transformations

This Appendix describes how ontologies of different versions of Java API packages are automatically generated. We chose to use model transformation to transform UML models of Java API into Java platform ontologies. For all JREs to be considered, the API, which is contained in a jar file bundled with the JRE, is reverse engineered as a UML model.

Each UML model that is reverse engineered in this way is transformed into a Java platform ontology. The following sections contain the ATL model transformations that we use. The "UML2Comparison.atl" library (see section A.3) is the most interesting part here, since it describes the meaning of compatibility and equivalence for models, packages and classifiers in OCL. In order to create an OWL ontology with a model transformation, a meta-model of OWL ontologies is needed. OMG has defined the Ontology Definition Metamodel (ODM) [OMG06c] exactly for this purpose. We use an EMF-based implementation of ODM[1].

Since we want to relate each Java API to other Java APIs, we need to add the ontologies of related APIs as input to the model transformation such that links between ontologies can be made. This requires a two-pass model transformation approach, in which the first version of each API ontology contains no relationships to other API ontologies. The second version of each API ontology can then use the first version of other API ontologies to describe its relationships. An Ant "build.xml" script[2] that implements this approach is provided in section A.5.

The latest version of the transformations and the build script can always be found at http://ssel.vub.ac.be/viewcvs/viewcvs.py/PlatformKit/platformkit-java/transformations/.

[1] http://www.alphaworks.ibm.com/tech/semanticstk

[2] Ant scripts are the Makefile equivalent for Java-based software development: they execute the different steps required for building and/or packaging software – or models is this case.

A.1 UML2ToPackageAPIOntology.atl

```
-- @atlcompiler atl2006
-- $Id: UML2ToPackageAPIOntology.atl 7062 2007-06-30 11:16:05Z dwagelaa $
-- Transforms a UML2 model to an API OWL ontology containing all the packages
module UML2ToPackageAPIOntology; --extends UML2ToAPIOntology

create OUT : OWL from IN : UML2, PLATFORM : OWL, JAVA : OWL;

uses UML2ToAPIOntology;

-- ===========================================================================
-- helper attributes begin
-- ===========================================================================

helper def : UML2ToPackageAPIOntologyVersionString : String =
  '$Id: UML2ToPackageAPIOntology.atl 7062 2007-06-30 11:16:05Z dwagelaa $';

-- ===========================================================================
-- helper attributes end
-- ===========================================================================

-- ===========================================================================
-- transformation rules begin
-- ===========================================================================

rule Model {
  from s : UML2!uml::Model (thisModule.inElements->includes(s))
  using {
    prevNotEmpty : Boolean = thisModule.prevModels
      ->select(m|m.packagedElement->notEmpty())->notEmpty();
    superClasses : Sequence(OWL!owl::OWLClass) = s.compatibleClasses
      ->select(c|s.equivalentClasses->excludes(c));
  }
  to n : OWL!rdfs::Namespace mapsTo s (
    URI <- 'http://ssel.vub.ac.be/platformkit/' + s.name + '#',
    name <- s.name),
     xsd : OWL!rdfs::Namespace (
      URI <- 'http://www.w3.org/2001/XMLSchema#',
    name <- 'xsd'),
     units : OWL!rdfs::Namespace (
      URI <- 'http://localhost/~dennis/Units.owl#',
    name <- 'units'),
     platform : OWL!rdfs::Namespace (
      URI <- 'http://localhost/~dennis/Platform.owl#',
    name <- 'platform'),
     java : OWL!rdfs::Namespace (
      URI <- 'http://localhost/~dennis/Java.owl#',
    name <- 'java'),
     ont : OWL!owl::OWLOntology (
      localName <- '',
    namespace <- n,
    RDFSLabel <- Sequence{label},
    RDFSComment <- Sequence{comment},
    ownedNamespace <- Set{n, xsd, units, platform, java}->union(
      if prevNotEmpty then thisModule.importedOntologies
        ->collect(o|thisModule.PrevNamespace(o))
      else Set{} endif),
-- !!! OWLImports generates bogus namespace !!!
    OWLImports <- thisModule.platformOntology->union(
      thisModule.javaOntology)->union(
      if prevNotEmpty then thisModule.importedOntologies
      else Set{} endif),
    contains <- Sequence{apiClass, jreClass}->union(
```

```
      thisModule.includedPackages)),
     label : OWL!rdfs::PlainLiteral (
      language <- 'en',
    lexicalForm <- s.name),
     comment : OWL!rdfs::PlainLiteral (
      language <- 'en',
    lexicalForm <- 'Generated by ' +
      thisModule.UML2ToPackageAPIOntologyVersionString + ' and ' +
      thisModule.UML2ToAPIOntologyVersionString + ' and ' +
      thisModule.UML2ComparisonVersionString),
     apiClass : OWL!owl::OWLClass (
      localName <- s.name.firstToUpper() + 'ClassLibrary',
    RDFSSubClassOf <- thisModule.includedPackages->union(superClasses),
    OWLEquivalentClass <- if s.equivalentClasses->isEmpty()
      then Sequence{} else s.equivalentClasses endif,
    RDFSComment <- Sequence{apiComment},
    namespace <- s),
     apiComment : OWL!rdfs::PlainLiteral (
      language <- 'en',
    lexicalForm <- 'JavaLibrary implementing the entire API for ' + s.name),
     jreClass : OWL!owl::OWLClass (
      localName <- s.name.firstToUpper(),
    RDFSSubClassOf <- Sequence{
      thisModule.javaJRE, builtinJavaLibraryRestriction},
    RDFSComment <- Sequence{jreComment},
    namespace <- s),
     jreComment : OWL!rdfs::PlainLiteral (
      language <- 'en',
    lexicalForm <- s.name + ' Java Runtime Environment'),
     builtinJavaLibraryRestriction : OWL!owl::SomeValuesFromRestriction (
      OWLOnProperty <- thisModule.jreProvidesBuiltinJavaLibrary,
    OWLSomeValuesFrom <- apiClass,
    namespace <- s)
}

rule Package {
  from s : UML2!uml::Package (thisModule.includedPackages->includes(s))
  using {
    superClasses : Sequence(OWL!owl::OWLClass) = s.compatibleClasses
      ->select(c|s.equivalentClasses->excludes(c));
  }
  to t : OWL!owl::OWLClass mapsTo s (
    localName <- (s.ontClassName + 'Library')
      .debug(thisModule.modelName.prefix + 'Package'),
    RDFSSubClassOf <- if superClasses->isEmpty()
      then thisModule.javaLibrary else superClasses endif,
    OWLEquivalentClass <- s.equivalentClasses,
    RDFSComment <- Sequence{comment},
    RDFSSeeAlso <- s.references,
    namespace <- s.getModel()),
     comment : OWL!rdfs::PlainLiteral (
      language <- 'en',
    lexicalForm <- 'JavaLibrary implementing the ' + s.javaQualifiedName +
      ' package for ' + thisModule.modelName)
}

-- ============================================================================
-- transformation rules end
-- ============================================================================
```

A.2 UML2ToAPIOntology.atl

```
-- @atlcompiler atl2006
```

```
-- $Id: UML2ToAPIOntology.atl 7062 2007-06-30 11:16:05Z dwagelaa $
-- Base module for transforming a UML2 model to an API OWL ontology
module UML2ToAPIOntology; --abstract

create OUT : OWL from IN : UML2, PLATFORM : OWL, JAVA : OWL;
  --, PREVOUT : OWL, PREVIN : UML2;

uses UML2Comparison;

-- ============================================================================
-- helper attributes begin
-- ============================================================================

helper def : UML2ToAPIOntologyVersionString : String =
  '$Id: UML2ToAPIOntology.atl 7062 2007-06-30 11:16:05Z dwagelaa $';

helper def : inElements : Set(UML2!ecore::EObject) =
  UML2!ecore::EObject.allInstancesFrom('IN');

helper def : includedPackages : Set(UML2!uml::Package) =
  UML2!uml::Package.allInstancesFrom('IN')
    ->select(p|p.oclIsTypeOf(UML2!uml::Package) and p.packagedElement
      ->select(c|c.oclIsKindOf(UML2!uml::Class) or
        c.oclIsKindOf(UML2!uml::Interface))->notEmpty());

helper def : platformPlatform : Set(OWL!owl::OWLClass) =
  OWL!owl::OWLClass.allInstancesFrom('PLATFORM')
    ->select(o|o.localName = 'Platform');

helper def : javaLibrary : Set(OWL!owl::OWLClass) =
  OWL!owl::OWLClass.allInstancesFrom('JAVA')
    ->select(o|o.localName = 'JavaLibrary');

helper def : javaJRE : Set(OWL!owl::OWLClass) =
  OWL!owl::OWLClass.allInstancesFrom('JAVA')
    ->select(o|o.localName = 'JRE');

helper def : platformProvidesFeature : OWL!owl::OWLObjectProperty =
  OWL!owl::OWLObjectProperty.allInstancesFrom('PLATFORM')
    ->select(p|p.localName = 'providesFeature')->first();

helper def : jreProvidesBuiltinJavaLibrary : OWL!owl::OWLObjectProperty =
  OWL!owl::OWLObjectProperty.allInstancesFrom('JAVA')
    ->select(p|p.localName = 'providesBuiltinJavaLibrary')->first();

helper def : platformOntology : Set(OWL!owl::OWLOntology) =
  OWL!owl::OWLOntology.allInstancesFrom('PLATFORM');

helper def : javaOntology : Set(OWL!owl::OWLOntology) =
  OWL!owl::OWLOntology.allInstancesFrom('JAVA');

helper def : importedOntologies : Set(OWL!owl::OWLOntology) =
  OWL!owl::OWLOntology.allInstances()->select(o|
    OWL!owl::OWLOntology.allInstancesFrom('PLATFORM')->excludes(o) and
    OWL!owl::OWLOntology.allInstancesFrom('JAVA')->excludes(o));

-- ============================================================================
-- general context helper attributes begin
-- ============================================================================

helper context UML2!uml::NamedElement def : javaQualifiedName : String =
  if self.owner.oclIsTypeOf(UML2!uml::Package)
  or self.owner.oclIsKindOf(UML2!uml::Classifier) then
    self.owner.javaQualifiedName + '.' + self.name
  else
```

```
    self.name
  endif;

helper context UML2!uml::NamedElement def : ontClassName : String =
  if self.owner.oclIsTypeOf(UML2!uml::Package) then
    self.owner.ontClassName
  else '' endif
  + self.name.firstToUpper();

-- ================================================================
-- helper attributes for finding references
-- ================================================================

helper context UML2!uml::Package def : references : Set(UML2!uml::Package) =
  self.allOwnedClassifiers
    ->collect(c|c.references)->flatten()->asSet()
    ->collect(r|r.referencesOtherPackageThan(self))->flatten()->asSet()
    .debug(thisModule.modelName.prefix + self.qualifiedName +
      ' referenced packages');

-- Non-transitive references
helper context UML2!uml::Classifier def : references :
Set(UML2!uml::Classifier) =
  self.general->union(
    self.suppliers->select(s|s.oclIsKindOf(UML2!uml::Classifier)))->union(
    self.feature->collect(f|f.referencesOtherThan(self)))
  ->flatten()->asSet();

-- ================================================================
-- helper attributes for determining compatibility
-- ================================================================

helper context UML2!uml::NamedElement def : owlClassesInPrev :
Sequence(OWL!owl::OWLClass) =
  OWL!owl::OWLClass.allInstances()
    ->select(c|c.localName = self.ontClassName + 'Library')
    ->select(o|o.namespace.name = self.getModel().name);

helper context UML2!uml::Package def : compatibleClasses :
Sequence(OWL!owl::OWLClass) =
  self.compatibleInPrev->collect(p|p.owlClassesInPrev)->flatten();

helper context UML2!uml::Package def : equivalentClasses :
Sequence(OWL!owl::OWLClass) =
  self.equivalentInPrev->collect(p|p.owlClassesInPrev)->flatten();

-- ================================================================
-- helper attributes end
-- ================================================================

-- ================================================================
-- general helper methods
-- ================================================================

helper context String def: firstToUpper() : String =
  self.substring(1, 1).toUpper() + self.substring(2, self.size());

-- ================================================================
-- helper methods for finding references
-- ================================================================

helper context UML2!uml::PackageableElement def : referencesOtherPackageThan
(p : UML2!uml::Package) : Set(UML2!uml::Package) =
  let np : UML2!uml::Package = self.getNearestPackage() in
    if np <> p then Set{np} else Set{} endif;
```

```
helper context UML2!uml::BehavioralFeature def : referencesOtherThan
(c : UML2!uml::Classifier) : Sequence(UML2!uml::Classifier) =
  self.ownedParameter->collect(p|p.referencesOtherThan(c))->flatten();

helper context UML2!uml::TypedElement def : referencesOtherThan
(c : UML2!uml::Classifier) : Sequence(UML2!uml::Classifier) =
  if self.type.oclIsKindOf(UML2!uml::Class)
  or self.type.oclIsKindOf(UML2!uml::Interface) then
    if self.type.getModel() = c.getModel() and self.type <> c
    then Sequence{self.type}
    else Sequence{} endif
  else Sequence{} endif;

-- ================================================================
-- helper methods end
-- ================================================================

-- ================================================================
-- transformation rules begin
-- ================================================================

rule PrevNamespace(o : OWL!owl::OWLOntology) {
  to n : OWL!rdfs::Namespace (
    URI <- o.namespace.URI,
    name <- o.namespace.name)
  do {
    n;
  }
}

-- ================================================================
-- transformation rules end
-- ================================================================
```

A.3 UML2Comparison.atl

```
-- @atlcompiler atl2006
-- $Id:UML2Comparison.atl 7084 2007-07-10 13:19:05Z dwagelaa $
-- Library for comparing a UML2 model to other versions of the UML2 model
library UML2Comparison;

-- ================================================================
-- helper attributes begin
-- ================================================================

helper def : UML2ComparisonVersionString : String =
  '$Id:UML2Comparison.atl 7084 2007-07-10 13:19:05Z dwagelaa $';

helper def : prevModels : Set(UML2!uml::Model) =
  UML2!uml::Model.allInstances()->select(m|
    UML2!uml::Model.allInstancesFrom('IN')->excludes(m));

helper def : prevPackages : Set(UML2!uml::Package) =
  UML2!uml::Package.allInstances()->select(p|
    UML2!uml::Package.allInstancesFrom('IN')->excludes(p));

helper def : modelName : String =
  UML2!uml::Model.allInstancesFrom('IN')->asSequence()->first().name;

-- ================================================================
-- general context helper attributes begin
-- ================================================================
```

```
helper context String def : prefix : String =
  '[' + self + '] ';

helper context UML2!uml::NamedElement def : umlQualifiedName : String =
  if self.oclIsKindOf(UML2!uml::Feature) then
    self.name
  else
    if self.owner.oclIsTypeOf(UML2!uml::Package)
    or self.owner.oclIsKindOf(UML2!uml::Classifier) then
      self.owner.umlQualifiedName + '::' + self.name
    else
      self.name
    endif
  endif;

helper context UML2!uml::Model def : allOwnedPackages :
Sequence(UML2!uml::Package) =
  self.allOwnedElements()
    ->select(o|o.oclIsKindOf(UML2!uml::Package));

helper context UML2!uml::Package def : allOwnedClassifiers :
Sequence(UML2!uml::Classifier) =
  self.allOwnedElements()
    ->select(o|o.oclIsKindOf(UML2!uml::Classifier))
    ->select(c|c.getNearestPackage() = self and not c.name.endsWith('[]'));

helper context UML2!uml::Model def : allOwnedPackages :
Sequence(UML2!uml::Package) =
  self.allOwnedElements()
    ->select(o|o.oclIsKindOf(UML2!uml::Package));

helper context UML2!uml::Package def : allOwnedClassifiers :
Sequence(UML2!uml::Classifier) =
  self.allOwnedElements()
    ->select(o|o.oclIsKindOf(UML2!uml::Classifier))
    ->select(c|c.getNearestPackage() = self and not c.name.endsWith('[]'));

helper context UML2!uml::Element def : myOclType : OclType =
  let type : OclType = self.oclType()
  in if type = UML2!uml::DataType
     then UML2!uml::Classifier
     else type endif;

-- ================================================================
-- helper attributes for determining compatibility
-- ================================================================

helper context UML2!uml::Interface def : suppliers :
Set(UML2!uml::NamedElement) =
  self.clientDependency->collect(d|d.supplier
      ->select(n|not n.name.oclIsUndefined())
    )->flatten()->asSet()
  ->union(self.general);

helper context UML2!uml::NamedElement def : suppliers :
Set(UML2!uml::NamedElement) =
  self.clientDependency->collect(d|d.supplier
      ->select(n|not n.name.oclIsUndefined())
    )->flatten()->asSet();

helper context UML2!uml::NamedElement def : allSuppliers :
Set(UML2!uml::NamedElement) =
  self.suppliers->union(
    self.suppliers->collect(s|s.allSuppliers)->flatten())->asSet();
```

```
helper context UML2!uml::NamedElement def : owlClassesInPrev :
Sequence(OWL!owl::OWLClass) =
  OWL!owl::OWLClass.allInstances()
    ->select(c|c.localName = self.ontClassName + 'Library')
    ->select(o|o.namespace.name = self.getModel().name);

helper context UML2!uml::Model def : compatibleInPrev :
Sequence(UML2!uml::Model) =
  thisModule.prevModels
    ->select(m|self.hasAllPackagesCompatibleWith(m))
    .debug(thisModule.modelName.prefix + self.qualifiedName +
      ' compatible models');

helper context UML2!uml::Model def : equivalentInPrev :
Sequence(UML2!uml::Model) =
  thisModule.prevModels
    ->select(m|self.hasAllPackagesEquivalentWith(m))
    .debug(thisModule.modelName.prefix + self.qualifiedName +
      ' equivalent models');

helper context UML2!uml::Package def : compatibleInPrev :
Sequence(UML2!uml::Package) =
  thisModule.prevPackages
    ->select(p|self.isCompatibleWith(p))
    .debug(thisModule.modelName.prefix + self.qualifiedName +
      ' compatible packages');

helper context UML2!uml::Package def : equivalentInPrev :
Sequence(UML2!uml::Package) =
  self.compatibleInPrev->select(p|p.isCompatibleWith(self))
    .debug(thisModule.modelName.prefix + self.qualifiedName +
      ' equivalent packages');

helper context UML2!uml::Classifier def : compatibleInPrev :
Sequence(UML2!uml::Classifier) =
  let packageName : String = self.getNearestPackage().umlQualifiedName in
    thisModule.prevPackages
      ->select(p|p.umlQualifiedName = packageName)
      ->collect(p1|p1.allOwnedClassifiers)->flatten()
      ->select(c|self.isCompatibleWith(c));

helper context UML2!uml::Classifier def : equivalentInPrev :
Sequence(UML2!uml::Classifier) =
  self.compatibleInPrev->select(c|c.isCompatibleWith(self));

helper context UML2!uml::Classifier def : allFeatures :
Set(UML2!uml::Feature) =
  self.feature
    ->union(self.general
      ->collect(c|c.allFeatures)->flatten()->asSet())
    ->union(self.suppliers
      ->select(s|s.oclIsKindOf(UML2!uml::Classifier))
      ->collect(c|c.allFeatures)->flatten()->asSet());

-- ========================================================================
-- helper attributes end
-- ========================================================================

-- ========================================================================
-- helper methods for determining compatibility
-- ========================================================================

helper context UML2!uml::Model def : hasAllPackagesCompatibleWith
(m : UML2!uml::Model) : Boolean =
```

```
  let compPackages : Sequence(UML2!uml::Package) = m.allOwnedPackages
    ->select(p|not self.hasPackageCompatibleWith(p))
  in if compPackages->isEmpty() then true
     else compPackages.debug(thisModule.modelName.prefix +
        self.qualifiedName + ' misses packages compatible with ')->isEmpty()
     endif;

helper context UML2!uml::Model def : hasPackageCompatibleWith
(p : UML2!uml::Package) : Boolean =
  self.allOwnedPackages
      ->select(sp|sp.compatibleInPrev->includes(p))->notEmpty();

helper context UML2!uml::Model def : hasAllPackagesEquivalentWith
(m : UML2!uml::Model) : Boolean =
  let equivPackages : Sequence(UML2!uml::Package) = m.allOwnedPackages
    ->select(p|not self.hasPackageEquivalentWith(p))
  in if equivPackages->isEmpty() then true
     else equivPackages.debug(thisModule.modelName.prefix +
        self.qualifiedName + ' misses packages equivalent with ')->isEmpty()
     endif;

helper context UML2!uml::Model def : hasPackageEquivalentWith
(p : UML2!uml::Package) : Boolean =
  self.allOwnedPackages
      ->select(sp|sp.equivalentInPrev->includes(p))->notEmpty();

helper context UML2!uml::Package def : isCompatibleWith
(p : UML2!uml::Package) : Boolean =
  if self.namedElementIsCompatibleWith(p) then
    let compClassifiers : Sequence(UML2!uml::Classifier) =
      p.allOwnedClassifiers->select(c|
        not self.hasOwnedClassifierCompatibleWith(c))
    in if compClassifiers->isEmpty() then true
       else compClassifiers
          .debug(thisModule.modelName.prefix + self.qualifiedName +
            ' misses classifiers compatible with ')
          ->isEmpty() endif
  else false endif;

helper context UML2!uml::Package def : hasOwnedClassifierCompatibleWith
(c : UML2!uml::Classifier) : Boolean =
  self.allOwnedClassifiers->select(o|o.isCompatibleWith(c))->notEmpty();

helper context UML2!uml::Classifier def : isCompatibleWith
(c : UML2!uml::Classifier) : Boolean =
  if self.namedElementIsCompatibleWith(c) then
    if (if self.isAbstract then c.isAbstract else true endif) then
      if (let compGenerals : Sequence(UML2!uml::Classifier) =
            c.general->select(g|not self.hasGeneralCompatibleWith(g))
          in if compGenerals->isEmpty() then true
             else compGenerals
                .debug(thisModule.modelName.prefix + self.qualifiedName +
                  ' misses generals compatible with ')
                ->isEmpty() endif)
      then (let compFeatures : Sequence(UML2!uml::Feature) =
             c.allFeatures->select(f|not self.hasFeatureCompatibleWith(f))
          in if compFeatures->isEmpty() then true
             else compFeatures
                .debug(thisModule.modelName.prefix + self.qualifiedName +
                  ' misses features compatible with ')
                ->isEmpty() endif)
      else false endif
    else false endif
  else false endif;
```

```
helper context UML2!uml::Classifier def : hasFeatureCompatibleWith
(f : UML2!uml::Feature) : Boolean =
  self.allFeatures->select(o|o.isCompatibleWith(f))->notEmpty();

helper context UML2!uml::StructuralFeature def : isCompatibleWith
(f : UML2!uml::StructuralFeature) : Boolean =
  if self.typedElementIsCompatibleWith(f)
  then (self.isStatic = f.isStatic) and
     (self.isReadOnly implies f.isReadOnly) and
     (self.isLeaf implies f.isLeaf)
  else false endif;

helper context UML2!uml::BehavioralFeature def : isCompatibleWith
(f : UML2!uml::BehavioralFeature) : Boolean =
  if self.namedElementIsCompatibleWith(f)
  then if (self.isStatic = f.isStatic)
       and (self.ownedParameter->size() = f.ownedParameter->size())
       and (self.isAbstract implies f.isAbstract)
       and (self.isLeaf implies f.isLeaf)
     then thisModule.compatibleParameters(self.ownedParameter,
        f.ownedParameter)
     else false endif
  else false endif;

helper def : compatibleParameters(selfpars : Sequence(UML2!uml::Parameter),
otherpars : Sequence(UML2!uml::Parameter)) : Boolean =
  let selfpar : UML2!uml::Parameter = selfpars->first() in
  let otherpar : UML2!uml::Parameter = otherpars->first() in
  if selfpar.oclIsUndefined() then otherpar.oclIsUndefined()
  else
    if selfpar.isCompatibleWith(otherpar) then
      thisModule.compatibleParameters(
        selfpars->excluding(selfpar),
        otherpars->excluding(otherpar))
    else false endif
  endif;

helper context UML2!uml::Parameter def : isCompatibleWith
(p : UML2!uml::Parameter) : Boolean =
  if p.oclIsUndefined() then false
  else
    if self.typedElementIsCompatibleWith(p)
    then (self.direction = p.direction)
    else false endif
  endif;

helper context UML2!uml::TypedElement def : isCompatibleWith
(t : UML2!uml::TypedElement) : Boolean =
  self.typedElementIsCompatibleWith(t);

helper context UML2!uml::NamedElement def : isCompatibleWith
(e : UML2!uml::NamedElement) : Boolean =
  self.namedElementIsCompatibleWith(e);

helper context UML2!uml::NamedElement def : visibilityIsCompatibleWith
(e : UML2!uml::NamedElement) : Boolean =
  if (self.visibility = #public) then true
  else
    if (self.visibility = #protected)
    then (e.visibility = #protected) or (e.visibility = #private)
    else
      if (self.visibility = #package)
      then (e.visibility = #package) or (e.visibility = #private)
      else (self.visibility = e.visibility) endif
    endif
```

```
  endif;

helper context UML2!uml::TypedElement def : typedElementIsCompatibleWith
(t : UML2!uml::TypedElement) : Boolean =
  if self.namedElementIsCompatibleWith(t) then
    if self.type.oclIsUndefined() then
      t.type.oclIsUndefined()
    else
      if t.type.oclIsUndefined() then false
      else (self.type.umlQualifiedName = t.type.umlQualifiedName) endif
    endif
  else false endif;

helper context UML2!uml::NamedElement def : namedElementIsCompatibleWith
(e : UML2!uml::NamedElement) : Boolean =
  if (self.umlQualifiedName = e.umlQualifiedName) then
  if (self.myOclType.conformsTo(e.myOclType))
  and (self.visibilityIsCompatibleWith(e)) then
      let compSuppliers : Sequence(UML2!uml::NamedElement) =
        e.suppliers->select(d|not self.hasSupplierCompatibleWith(d))
      in if compSuppliers->isEmpty() then true
         else compSuppliers
            .debug(thisModule.modelName.prefix + self.qualifiedName +
              ' misses dependency suppliers compatible with ')
            ->isEmpty() endif
    else false endif
  else false endif;

helper context UML2!uml::Classifier def : hasGeneralCompatibleWith
(c : UML2!uml::Classifier) : Boolean =
  self.general->select(g|
    if (g.umlQualifiedName = c.umlQualifiedName) then true
    else g.hasGeneralCompatibleWith(c) endif)->notEmpty();

helper context UML2!uml::Classifier def : hasSupplierCompatibleWith
(e : UML2!uml::NamedElement) : Boolean =
  if self.namedElementHasSupplierCompatibleWith(e) then true
  else self.general->select(g|g.hasSupplierCompatibleWith(e))->notEmpty()
  endif;

helper context UML2!uml::NamedElement def : hasSupplierCompatibleWith
(e : UML2!uml::NamedElement) : Boolean =
  self.namedElementHasSupplierCompatibleWith(e);

helper context UML2!uml::NamedElement def :
namedElementHasSupplierCompatibleWith(e : UML2!uml::NamedElement) : Boolean =
  self.allSuppliers
    ->select(s|s.umlQualifiedName = e.umlQualifiedName)->notEmpty();

-- ==========================================================================
-- helper methods end
-- ==========================================================================
```

A.4 UML2CompatibilityComparison.atl

```
-- @atlcompiler atl2006
-- $Id: UML2CompatibilityComparison.atl 7084 2007-07-10 13:19:05Z dwagelaa $
-- Query for determining the compatibility of a UML2 model to other
-- versions of the UML2 model
query UML2CompatibilityComparison = UML2!uml::Model.allInstancesFrom('IN')
  ->collect(m|m.compatibleInPrev)->flatten()->notEmpty()
  .debug('Model is compatible with previous model');
```

```
uses UML2Comparison;
```

A.5 Parallel build script

```
<project name="platformkit-java/transformations" basedir="." default="transform">

    <macrodef name="preUML2ToAPIOntology">
        <attribute name="in" default="MODEL.IN" />
        <attribute name="out" default="MODEL.OUT" />
        <attribute name="path" default="MODEL.OUT.PATH" />
        <sequential>
     <am3.atl path="/${ant.project.name}/UML2ToAPIOntology.atl"
              allowInterModelReferences="true">
     <inmodel name="UML2" model="UML2"/>
     <inmodel name="OWL" model="OWL"/>
     <inmodel name="IN" model="@{in}"/>
     <inmodel name="PLATFORM" model="Platform"/>
     <inmodel name="JAVA" model="Java"/>
     <outmodel name="OUT" model="@{out}" metamodel="OWL" path="@{path}"/>
     <superimpose path="/${ant.project.name}/UML2ToPackageAPIOntology.asm"/>
     <library name="UML2Comparison" path="/${ant.project.name}/UML2Comparison.asm"/>
     </am3.atl>
        </sequential>
    </macrodef>

    <macrodef name="UML2ToAPIOntology">
        <attribute name="in" default="MODEL.IN" />
        <attribute name="out" default="MODEL.OUT" />
        <attribute name="path" default="MODEL.OUT.PATH" />
        <sequential>
     <am3.atl path="/${ant.project.name}/UML2ToAPIOntology.atl"
              allowInterModelReferences="true">
     <inmodel name="UML2" model="UML2"/>
     <inmodel name="OWL" model="OWL"/>
     <inmodel name="IN" model="@{in}"/>
     <inmodel name="PLATFORM" model="Platform"/>
     <inmodel name="JAVA" model="Java"/>
     <outmodel name="OUT" model="@{out}" metamodel="OWL" path="@{path}"/>
     <superimpose path="/${ant.project.name}/UML2ToPackageAPIOntology.asm"/>
     <library name="UML2Comparison" path="/${ant.project.name}/UML2Comparison.asm"/>
     </am3.atl>
         <am3.saveModel model="@{out}" path="@{path}"/>
        </sequential>
    </macrodef>

    <macrodef name="UML2ToAPIOntology1">
        <attribute name="in" default="MODEL.IN" />
        <attribute name="prevout" default="MODEL.PREV.OUT" />
        <attribute name="previn" default="MODEL.PREV.IN" />
        <attribute name="out" default="MODEL.OUT" />
        <attribute name="path" default="MODEL.OUT.PATH" />
        <sequential>
     <am3.atl path="/${ant.project.name}/UML2ToAPIOntology.atl"
              allowInterModelReferences="true">
     <inmodel name="UML2" model="UML2"/>
     <inmodel name="OWL" model="OWL"/>
     <inmodel name="IN" model="@{in}"/>
     <inmodel name="PLATFORM" model="Platform"/>
     <inmodel name="JAVA" model="Java"/>
     <inmodel name="PREVOUT" model="@{prevout}"/>
     <inmodel name="PREVIN" model="@{previn}"/>
     <outmodel name="OUT" model="@{out}" metamodel="OWL" path="@{path}"/>
     <superimpose path="/${ant.project.name}/UML2ToPackageAPIOntology.asm"/>
     <library name="UML2Comparison" path="/${ant.project.name}/UML2Comparison.asm"/>
```

```
 </am3.atl>
     <am3.saveModel model="@{out}" path="@{path}"/>
    </sequential>
</macrodef>

<macrodef name="UML2ToAPIOntology2">
    <attribute name="in" default="MODEL.IN" />
    <attribute name="prevout" default="MODEL.PREV.OUT" />
    <attribute name="previn" default="MODEL.PREV.IN" />
    <attribute name="prevout2" default="MODEL.PREV.OUT2" />
    <attribute name="previn2" default="MODEL.PREV.IN2" />
    <attribute name="out" default="MODEL.OUT" />
    <attribute name="path" default="MODEL.OUT.PATH" />
    <sequential>
 <am3.atl path="/${ant.project.name}/UML2ToAPIOntology.atl"
          allowInterModelReferences="true">
 <inmodel name="UML2" model="UML2"/>
 <inmodel name="OWL" model="OWL"/>
 <inmodel name="IN" model="@{in}"/>
 <inmodel name="PLATFORM" model="Platform"/>
 <inmodel name="JAVA" model="Java"/>
 <inmodel name="PREVOUT" model="@{prevout}"/>
 <inmodel name="PREVIN" model="@{previn}"/>
 <inmodel name="PREVOUT2" model="@{prevout2}"/>
 <inmodel name="PREVIN2" model="@{previn2}"/>
 <outmodel name="OUT" model="@{out}" metamodel="OWL" path="@{path}"/>
 <superimpose path="/${ant.project.name}/UML2ToPackageAPIOntology.asm"/>
 <library name="UML2Comparison" path="/${ant.project.name}/UML2Comparison.asm"/>
 </am3.atl>
     <am3.saveModel model="@{out}" path="@{path}"/>
    </sequential>
</macrodef>

<macrodef name="UML2ToAPIOntology3">
    <attribute name="in" default="MODEL.IN" />
    <attribute name="prevout" default="MODEL.PREV.OUT" />
    <attribute name="previn" default="MODEL.PREV.IN" />
    <attribute name="prevout2" default="MODEL.PREV.OUT2" />
    <attribute name="previn2" default="MODEL.PREV.IN2" />
    <attribute name="prevout3" default="MODEL.PREV.OUT3" />
    <attribute name="previn3" default="MODEL.PREV.IN3" />
    <attribute name="out" default="MODEL.OUT" />
    <attribute name="path" default="MODEL.OUT.PATH" />
    <sequential>
 <am3.atl path="/${ant.project.name}/UML2ToAPIOntology.atl"
          allowInterModelReferences="true">
 <inmodel name="UML2" model="UML2"/>
 <inmodel name="OWL" model="OWL"/>
 <inmodel name="IN" model="@{in}"/>
 <inmodel name="PLATFORM" model="Platform"/>
 <inmodel name="JAVA" model="Java"/>
 <inmodel name="PREVOUT" model="@{prevout}"/>
 <inmodel name="PREVIN" model="@{previn}"/>
 <inmodel name="PREVOUT2" model="@{prevout2}"/>
 <inmodel name="PREVIN2" model="@{previn2}"/>
 <inmodel name="PREVOUT3" model="@{prevout3}"/>
 <inmodel name="PREVIN3" model="@{previn3}"/>
 <outmodel name="OUT" model="@{out}" metamodel="OWL" path="@{path}"/>
 <superimpose path="/${ant.project.name}/UML2ToPackageAPIOntology.asm"/>
 <library name="UML2Comparison" path="/${ant.project.name}/UML2Comparison.asm"/>
 </am3.atl>
     <am3.saveModel model="@{out}" path="@{path}"/>
    </sequential>
</macrodef>
```

```
<target name="loadModels">
<am3.loadModel modelHandler="EMF" name="OWL"
               metamodel="MOF"    nsuri="http:///org/eclipse/owl.ecore"/>
<am3.loadModel modelHandler="EMF" name="Platform"
               metamodel="OWL"    path="/platformkit-java/Platform.owl"/>
<am3.loadModel modelHandler="EMF" name="Java"
               metamodel="OWL"    path="/platformkit-java/Java.owl"/>
<am3.loadModel modelHandler="EMF" name="EmptyOWL"
               metamodel="OWL"    path="/platformkit-java/empty.owl"/>
<am3.loadModel modelHandler="EMF" name="UML2"
               metamodel="MOF"    nsuri="http://www.eclipse.org/uml2/2.0.0/UML"/>
<am3.loadModel modelHandler="EMF" name="EmptyUML"
               metamodel="UML2"   path="/platformkit-java/empty.uml"/>
<am3.loadModel modelHandler="EMF" name="J2ME-MIDP-1_0"
               metamodel="UML2"   path="/platformkit-java/j2me-midp-1_0-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2ME-MIDP-2_0"
               metamodel="UML2"   path="/platformkit-java/j2me-midp-2_0-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2ME-PP-1_0"
               metamodel="UML2"   path="/platformkit-java/j2me-pp-1_0-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2ME-PP-1_1"
               metamodel="UML2"   path="/platformkit-java/j2me-pp-1_1-api.uml"/>
<am3.loadModel modelHandler="EMF" name="PJ-1_1"
               metamodel="UML2"   path="/platformkit-java/personaljava-1_1-api.uml"/>
<am3.loadModel modelHandler="EMF" name="JDK-1_1"
               metamodel="UML2"   path="/platformkit-java/jdk-1_1-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2SE-1_2"
               metamodel="UML2"   path="/platformkit-java/j2se-1_2-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2SE-1_3"
               metamodel="UML2"   path="/platformkit-java/j2se-1_3-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2SE-1_4"
               metamodel="UML2"   path="/platformkit-java/j2se-1_4-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2SE-1_5"
               metamodel="UML2"   path="/platformkit-java/j2se-1_5-api.uml"/>
<am3.loadModel modelHandler="EMF" name="J2SE-1_6"
               metamodel="UML2"   path="/platformkit-java/j2se-1_6-api.uml"/>
</target>

<target name="pre-transform" depends="loadModels">
<nice currentpriority="curpri" newpriority="1"/>
<parallel threadCount="8"><!-- start heaviest task first -->
<preUML2ToAPIOntology in="J2SE-1_5"      out="preJ2SE-1_5_OWL"
                      path="/platformkit-java/j2se-1_5-api.owl"/>
<preUML2ToAPIOntology in="J2SE-1_4"      out="preJ2SE-1_4_OWL"
                      path="/platformkit-java/j2se-1_4-api.owl"/>
<preUML2ToAPIOntology in="J2SE-1_3"      out="preJ2SE-1_3_OWL"
                      path="/platformkit-java/j2se-1_3-api.owl"/>
<preUML2ToAPIOntology in="J2SE-1_2"      out="preJ2SE-1_2_OWL"
                      path="/platformkit-java/j2se-1_2-api.owl"/>
<preUML2ToAPIOntology in="JDK-1_1"       out="preJDK-1_1_OWL"
                      path="/platformkit-java/jdk-1_1-api.owl"/>
<preUML2ToAPIOntology in="PJ-1_1"        out="prePJ-1_1_OWL"
                      path="/platformkit-java/personaljava-1_1-api.owl"/>
<preUML2ToAPIOntology in="J2ME-PP-1_1"   out="preJ2ME-PP-1_1_OWL"
                      path="/platformkit-java/j2me-pp-1_1-api.owl"/>
<preUML2ToAPIOntology in="J2ME-PP-1_0"   out="preJ2ME-PP-1_0_OWL"
                      path="/platformkit-java/j2me-pp-1_0-api.owl"/>
<preUML2ToAPIOntology in="J2ME-MIDP-2_0" out="preJ2ME-MIDP-2_0_OWL"
                      path="/platformkit-java/j2me-midp-2_0-api.owl"/>
<UML2ToAPIOntology    in="J2ME-MIDP-1_0" out="J2ME-MIDP-1_0_OWL"
                      path="/platformkit-java/j2me-midp-1_0-api.owl"/>
</parallel>
<nice newpriority="${curpri}"/>
</target>

<target name="transform" depends="pre-transform">
```

```
<nice currentpriority="curpri" newpriority="1"/>
<parallel threadCount="8"><!-- start heaviest task first -->
<UML2ToAPIOntology1 in="J2SE-1_6"                  out="J2SE-1_6_OWL"
                    prevout="preJ2SE-1_5_OWL"      previn="J2SE-1_5"
                    path="/platformkit-java/j2se-1_6-api.owl"/>
<UML2ToAPIOntology1 in="J2SE-1_5"                  out="J2SE-1_5_OWL"
                    prevout="preJ2SE-1_4_OWL"      previn="J2SE-1_4"
                    path="/platformkit-java/j2se-1_5-api.owl"/>
<UML2ToAPIOntology2 in="J2SE-1_4"                  out="J2SE-1_4_OWL"
                    prevout="preJ2SE-1_3_OWL"      previn="J2SE-1_3"
                    prevout2="preJ2ME-PP-1_1_OWL"  previn2="J2ME-PP-1_1"
                    path="/platformkit-java/j2se-1_4-api.owl"/>
<UML2ToAPIOntology2 in="J2SE-1_3"                  out="J2SE-1_3_OWL"
                    prevout="preJ2SE-1_2_OWL"      previn="J2SE-1_2"
                    prevout2="preJ2ME-PP-1_0_OWL"  previn2="J2ME-PP-1_0"
                    path="/platformkit-java/j2se-1_3-api.owl"/>
<UML2ToAPIOntology3 in="J2SE-1_2"                  out="J2SE-1_2_OWL"
                    prevout="preJDK-1_1_OWL"       previn="JDK-1_1"
                    prevout2="prePJ-1_1_OWL"       previn2="PJ-1_1"
                    prevout3="preJ2ME-MIDP-2_0_OWL" previn3="J2ME-MIDP-2_0"
                    path="/platformkit-java/j2se-1_2-api.owl"/>
<UML2ToAPIOntology1 in="JDK-1_1"                   out="JDK-1_1_OWL"
                    prevout="prePJ-1_1_OWL"        previn="PJ-1_1"
                    path="/platformkit-java/jdk-1_1-api.owl"/>
<UML2ToAPIOntology1 in="PJ-1_1"                    out="PJ-1_1_OWL"
                    prevout="preJ2ME-MIDP-2_0_OWL" previn="J2ME-MIDP-2_0"
                    path="/platformkit-java/personaljava-1_1-api.owl"/>
<UML2ToAPIOntology2 in="J2ME-PP-1_1"               out="J2ME-PP-1_1_OWL"
                    prevout="preJ2SE-1_3_OWL"      previn="J2SE-1_3"
                    prevout2="preJ2ME-PP-1_0_OWL"  previn2="J2ME-PP-1_0"
                    path="/platformkit-java/j2me-pp-1_1-api.owl"/>
<UML2ToAPIOntology3 in="J2ME-PP-1_0"               out="J2ME-PP-1_0_OWL"
                    prevout="preJ2SE-1_2_OWL"      previn="J2SE-1_2"
                    prevout2="J2ME-MIDP-1_0_OWL"   previn2="J2ME-MIDP-1_0"
                    prevout3="preJ2ME-MIDP-2_0_OWL" previn3="J2ME-MIDP-2_0"
                    path="/platformkit-java/j2me-pp-1_0-api.owl"/>
<UML2ToAPIOntology1 in="J2ME-MIDP-2_0"             out="J2ME-MIDP-2_0_OWL"
                    prevout="J2ME-MIDP-1_0_OWL"    previn="J2ME-MIDP-1_0"
                    path="/platformkit-java/j2me-midp-2_0-api.owl"/>
</parallel>
<nice newpriority="${curpri}"/>
</target>

</project>
```

Appendix B

ConstraintSet sorting algorithm

This appendix lists the Java implementation of the sorting algorithm applied to a list of "ConstraintSet" elements in a PlatformKit model. "ConstraintSet" elements represent intersections of platform constraints for the purpose of sorting. The sorting algorithm is based on an OWL class hierarchy representation of the "ConstraintSet" elements and a pre-sorted list of the same "ConstraintSet" elements. The leaves in the class hierarchy are considered as *most-specific* and the roots as *least-specific*. This forms the basis for sorting *most-specific-first* or *least-specific-first*. The class hierarchy constitutes a partial ordering where each pair of "ConstraintSet" elements with a child-ancestor relationship is ordered and each pair without such a relationship (i.e. siblings) are not ordered. The algorithm attempts to preserve as much of the order of the pre-sorted list as possible, while enforcing the partial ordering given by the OWL class hierarchy.

The algorithm is implemented by two Java classes: TreeSorter and HierarchyComparator. The TreeSorter class takes a pre-sorted list as input and uses a HierarchyComparator instance to determine the partial ordering. TreeSorter outputs an updated list that conforms to the partial order prescribed by the HierarchyComparator.

B.1 TreeSorter.java

```
package be.ac.vub.platformkit.util;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Logger;

import junit.framework.Assert;
import be.ac.vub.platformkit.kb.Ontologies;
```

```
/**
 * Sorts the list by repeatedly removing the first smallest element
 * ("root" element). Can deal with incomparable elements (partially
 * ordered lists), since it only looks for elements that are guaranteed
 * smaller than the current element when comparing.
 * Preserves existing list order where possible.
 *
 * In Java, the Arrays.sort() methods use mergesort or a tuned
 * quicksort depending on the datatypes and for implementation
 * efficiency switch to insertion sort when fewer than seven array
 * elements are being sorted. These algorithms are only
 * applicable to totally ordered lists, however.
 *
 * @author dennis
 */
public class TreeSorter {
    private Logger logger = Logger.getLogger(Ontologies.LOGGER);
    private Comparator comp;

    /**
     * Creates a TreeSorter.
     * @param comp
     * @throws IllegalArgumentException if comp is null
     */
    public TreeSorter(Comparator comp)
    throws IllegalArgumentException {
        if (comp == null) {
            throw new IllegalArgumentException("Null␣comparator");
        }
        this.comp = comp;
    }

    /**
     * Sorts the list by repeatedly removing the smallest elements
     * ("root" elements).
     * @param list
     */
    public void sort(List list) {
        List sorted = new ArrayList();
        while (!list.isEmpty()) {
            Object removed = removeRootElement(list);
            Assert.assertNotNull(
                "Remove␣at␣least␣one␣element␣==␣false",
                removed);
            sorted.add(removed);
        }
        list.addAll(sorted);
    }

    /**
     * Removes the root (i.e. smallest) element from the list.
     * @param list
     * @return the root element.
     */
    private Object removeRootElement(List list) {
        for (Iterator ls = list.iterator(); ls.hasNext();) {
            Object element = ls.next();
            if (isRootElement(element, list)) {
                ls.remove();
                logger.info("Root␣element␣removed:␣" + element);
                return element;
            }
        }
        return null;
    }
```

```
    /**
     * @param obj
     * @param list
     * @return True if obj is a "root" element in list.
     */
    private boolean isRootElement(Object obj, List list) {
        for (Iterator ls = list.iterator(); ls.hasNext();) {
            Object element = ls.next();
            try {
                if (comp.compare(obj, element) > 0) {
                    logger.fine(obj + " not root; greater than " + element);
                    return false;
                }
            } catch (ClassCastException e) {
                logger.fine(obj + " and " + element + " not comparable");
            }
        }
        return true;
    }
}
```

B.2 HierarchyComparator.java

```
package be.ac.vub.platformkit.util;

import java.util.Comparator;
import java.util.logging.Logger;

import junit.framework.Assert;

import be.ac.vub.platformkit.Constraint;
import be.ac.vub.platformkit.kb.Ontologies;

import com.hp.hpl.jena.ontology.OntClass;

/**
 * Compares {@link Constraint} objects, such that the more specific (subclass)
 * constraint is considered smaller (MOST_SPECIFIC_FIRST) or greater
 * (LEAST_SPECIFIC_FIRST) than the less specific (superclass) constraint.
 * If both constraints are equivalent, they are considered equally specific.
 * If no subclass relationship can be determined,
 * an exception is thrown.
 * @author dennis
 */
public class HierarchyComparator implements Comparator {
    public static final int MOST_SPECIFIC_FIRST = -1;
    public static final int LEAST_SPECIFIC_FIRST = 1;

    private Logger logger = Logger.getLogger(Ontologies.LOGGER);
    private int mode;

    /**
     * Creates a HierarchyComparator.
     * @param mode MOST_SPECIFIC_FIRST or LEAST_SPECIFIC_FIRST.
     * @throws IllegalArgumentException if illegal mode is given.
     */
    public HierarchyComparator(int mode)
    throws IllegalArgumentException {
        super();
        if (Math.abs(mode) != 1) {
            throw new IllegalArgumentException("Invalid mode: " + mode);
        }
```

```
        this.mode = mode;
    }

    /**
     * @see Comparator#compare(T, T)
     * @throws ClassCastException if something else than Constraint objects
     * are compared or if no order can be determined.
     */
    public int compare(Object arg0, Object arg1)
    throws ClassCastException {
        OntClass c0 = ((Constraint) arg0).getOntClass();
        OntClass c1 = ((Constraint) arg1).getOntClass();
        Assert.assertNotNull(c0);
        Assert.assertNotNull(c1);
        if (c0.equals(c1) || c0.hasEquivalentClass(c1)) {
            logger.fine(c0 + "␣equivalent␣to␣" + c1);
            return 0;
        }
        if (c0.hasSuperClass(c1)) {
            logger.fine(c0 + "␣subclass␣of␣" + c1);
            return 1 * mode;
        }
        if (c0.hasSubClass(c1)) {
            logger.fine(c0 + "␣superclass␣of␣" + c1);
            return -1 * mode;
        }
        logger.fine(c0 + "␣orthogonal␣to␣" + c1);
        throw new ClassCastException(
            "Cannot␣determine␣order␣for␣" + c0 + "␣and␣" + c1);
    }

    /**
     * @see Comparator#equals(java.lang.Object)
     */
    public boolean equals(Object obj) {
        if (obj instanceof HierarchyComparator) {
            return ((HierarchyComparator) obj).mode == mode;
        } else {
            return false;
        }
    }

}
```

Appendix C

Example index page for PlatformKit deployment

This appendix lists an example index page for the deployment website of the instant messenger case study. The purpose of this index page is to invoke the PlatformKit Servlet with the correct parameters. The index page needs to be adapted with the correct URL of the PlatformKit Servlet and the correct URL of the ".platformkit" deployment model.

C.1 index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>PlatformKit - Instant Messenger</title>
</head>
<body>
  <h1>Instant Messaging Client Configuration</h1>
  <script language="JavaScript">
  <!-- //
    document.write(
      "<form name=\"config\" method=\"post\" action=\"http://" +
      location.hostname + ":8080/platformkitservlet/servlet/options\" " +
      "enctype=\"multipart/form-data\" onSubmit=\"startCounter();\">");
    document.write(
      "<input name=\"baseurl\" type=\"hidden\" value=\"" + location +
      "InstantMessengerDeployment.platformkit\"/>");
  //-->
  </script>
  <input name="result" type="hidden" value="mostspecific" />
  <input name="noValidate" type="hidden" value="false" />
  <table border="0" cellpadding="2" cellspacing="0">
  <tr>
    <td>Platform ontology: </td>
    <td>
      <input name="context" type="file"
        alt="Specify your OWL platform description here" />
    </td>
  </tr>
  <tr>
```

```
    <td></td>
    <td>
      <input type="submit" alt="Send your request to the reasoner" />
      <input name="counter" type="text" size="3" value="0" />
    </td>
  </tr>
  </table>
  </form>

  <script language="JavaScript">
  <!-- //
    document.write(
      "<p><a href=\"http://" + location.hostname +
      ":8080/platformkitservlet/servlet/options?baseurl=" + location +
      "InstantMessengerDeployment.platformkit" +
      "&result=mostspecific&noValidate=false\">");
    document.write(
      "Let the server auto-detect your platform</a>.</p>");
  //-->
  </script>

  <h2>Example platform descriptions</h2>
  <ul>
    <li><a href="example/Generic/JDK1.1PC.owl">
      Generic PC with JDK 1.1</a></li>
    <li><a href="example/Generic/JDK1.2PC.owl">
      Generic PC with JDK 1.2</a></li>
    <li><a href="example/Generic/JDK1.3PC.owl">
      Generic PC with JDK 1.3</a></li>
    <li><a href="example/Generic/JDK1.4PC.owl">
      Generic PC with JDK 1.4</a></li>
    <li><a href="example/Generic/JDK1.5PC.owl">
      Generic PC with JDK 1.5</a></li>
    <li><a href="example/Generic/JDK1.6PC.owl">
      Generic PC with JDK 1.6</a></li>
    <li><a href="example/Generic/PersonalJava1.1PocketPC.owl">
      Microsoft PocketPC with Personal Java 1.1</a></li>
    <li><a href="example/Generic/J2MEPP1.0PocketPC.owl">
      Microsoft PocketPC with J2ME PP 1.0</a></li>
    <li><a href="example/Generic/J2MEPP1.1PocketPC.owl">
      Microsoft PocketPC with J2ME PP 1.1</a></li>
    <li><a href="example/Generic/J2MEMIDP1.0Phone.owl">
      Mobile phone with J2ME MIDP 1.0</a></li>
    <li><a href="example/Generic/J2MEMIDP2.0Phone.owl">
      Mobile phone with J2ME MIDP 2.0</a></li>
    <li><a href="example/Sharp/ZaurusSL-C1000PP.owl">
      Sharp Zaurus SL-C1000 PDA with J2ME PP</a></li>
    <li><a href="example/Sharp/ZaurusSL-C1000Jeode.owl">
      Sharp Zaurus SL-C1000 PDA with Jeode Personal Java</a></li>
    <li><a href="example/Siemens/CX70v.owl">
      Siemens CX70v mobile phone with MIDP 2.0</a></li>
  </ul>
</body>
<script language="JavaScript">
<!-- //
  function startCounter() {
    document.config.counter.type = "text";
    setTimeout("doCount(0)", 1000);
  }

  function doCount(count) {
    count++;
    document.config.counter.value = count;
    setTimeout("doCount(" + count + ")", 1000);
  }
```

```
  document.config.counter.type = "hidden";
//-->
</script>
</html>
```

Bibliography

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In MoDELS 2006 [MoD06], pages 692–706. 6.6

[ADvSP04] João Paulo A. Almeida, Remco M. Dijkman, Marten van Sinderen, and Luís Ferreira Pires. On the notion of abstract platform in mda development. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC 2004), Monterey, CA, USA*, pages 253–263. IEEE Computer Society, September 2004. 2.2.4, 4.8

[BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortéz. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, December 2006. 5.3.1, 6.2, 6.6

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Application*. Cambridge University Press, Cambridge, UK, January 2003. 1.3.1, 3.1, 4.7.2

[BDD+05] Jean Bézivin, Vladan Devedžić, Dragan Djurić, J.M. Favreau, Dragan Gašević, and Frédéric Jouault. An M3-neutral infrastructure for bridging model engineering and ontology engineering. In *Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05) Geneva, Switzerland*, pages 159–171. Springer-Verlag, February 2005. 4.8

[BG04] Dan Brickley and R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium, Febru-

ary 2004. W3C Recommendation 10 February 2004, [Online] http://www.w3.org/TR/rdf-schema/. 3.1

[Bos06] Jan Bosch. The challenges of broadening the scope of software product families. *Communications of the ACM*, 49(12):41–44, December 2006. 1.3.2

[BRCTS06] David Benavides, Antonio Ruiz-Cortéz, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *Proceedings of Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06), Sitges, Spain*, October 2006. 5.3.1, 6.5.2, 6.6

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, August 2003. (document), 1.4, 2.3, 2.3.2, 2.9

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004. 1.1.1, 5.4

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortéz. Automated reasoning on feature models. In Oscar Pastor and Jo ao Falcão e Cunha, editors, *Proceeding of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer-Verlag, June 2005. 5.3.1, 6.6

[CDZ04] Gerardo Canfora, Giuseppe Di Santo, and Eugenio Zimeo. Toward seamless migration of Java AWT-based applications to personal wireless devices. In *Proceedings of the 11th Working Conference on Reverse Engineering (RE 2004), Delft, The Netherlands*, pages 38–47. IEEE Computer Society, November 2004. 1.1.1

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison Wesley, June 2000. 1.3.2, 5.3

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006. 8.5.3

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Proceedings of the Third Software Product-Line Conference (SPLC 2004), Boston, MA,*

USA, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer-Verlag, August 2004. 5.1

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, March 2005. Special Issue on Software Variability: Process and Management. 5.3

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, April 2005. Special Issue on Software Product Lines. 5.1, 5.3, 5.3, 5.4, 5.4.1, 6.6

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998. 5.1, 5.2

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison Wesley Professional, August 2001. 1.3.2, 5.1

[DBS+01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Scenarios for ambient intelligence in 2010. Technical report, IST Advisory Group (ISTAG), February 2001. [Online] ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf. 1.1

[DK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002. 5.1, 5.4, 5.4.1, 6.6

[GJJB00] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java™ Language Specification*. Prentice Hall, second edition, June 2000. 1.1

[GJJB05] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java™ Language Specification*. Prentice Hall, third edition, June 2005. 1.1

[Gru93] Tom R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993. 3.1

[HPS04] Ian Horrocks and Peter F. Patel-Schneider. Reducing owl entailment to description logic satisfiability. *Journal of Web Semantics*, 1(4):345–357, October 2004. 4.7.2

[HSGP06] Brian Henderson-Sellers and Cesar Gonzalez-Perez. Uses and abuses of the stereotype mechanism in uml 1.x and 2.0. In MoDELS 2006 [MoD06], pages 16–26. 2.3.3

[JK06] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006), Dijon, France*, April 2006. 2.4.2

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990. 5.3

[KKK+06] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In MoDELS 2006 [MoD06], pages 528–542. 4.8

[Kru06] Charles W. Krueger. New methods in software product line practice. *Communications of the ACM*, 49(12):37–40, December 2006. 1.3.2, 5.4, 6.3

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Addison Wesley Professional, 2003. (document), 2.1

[LBM+01] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001. 1.3.2, 5.4

[Man02] Mike Mannion. Using first-order logic for product line model validation. In *Proceedings of Software Product Lines : Second International Conference, SPLC 2, San Diego, CA, USA*, volume 2379 of *Lecture Notes in Computer Science*, pages 149–202. Springer-Verlag, August 2002. 5.3.1

[MC99] Luis Mandel and María Victoria Cengarle. On the expressive power of ocl. In *Proceedings of Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*

(FM'99), Toulouse, France, Volume I, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999. 6.6

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006. 8.5.3

[MH03] Ralf Möller and Volker Haarslev. Description logics for the semantic web: Racer as a basis for building agent systems. *Künstliche Intelligenz*, 17(3):10–15, July 2003. 1.3.1, 4.2, 7.1

[MKDS03] Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, and Beth Stearns. *Applying Enterprise JavaBeans*. Addison Wesley Professional, May 2003. 4.3

[MM03] Joaquin Miller and Jishnu Mukerji. *MDA Guide*. Object Management Group, Inc., June 2003. Version 1.0.1, omg/03-06-01. 1.1, 2.1, 2.2

[MoD06] *Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2006. C.1

[MRZ+06] Wen Jun Meng, Jürgen Rilling, Yonggang Zhang, René Witte, and Philippe Charland. An ontological software comprehension process model. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, Italy*, October 2006. 4.8

[MS96] Microsoft Corporation. *DCOM Technical Overview*, November 1996. [Online] http://msdn2.microsoft.com/en-us/library/ms809340.aspx. 4.3

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electr. Notes Theor. Comput. Sci.*, 127(3):113–128, April 2005. 1.1.1, 6.2

[Obj05] Object Management Group, Inc. *OCL 2.0 Specification*, June 2005. Version 2.0, ptc/2005-06-06. 2.4

[Old05] Jon Oldevik. Transformation composition modelling framework. In Lea Kutvonen and Nancy Alonistioti, editors, *Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2005), Athens, Greece*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer-Verlag, May 2005. 6.6

[OMG02] Object Management Group, Inc. *UML Profile for CORBA Specification*, April 2002. Version 1.0, formal/02-04-01. 2.2.4

[OMG04a] Object Management Group, Inc. *Common Object Request Broker Architecture: Core Specification*, March 2004. Version 3.0.3, formal/04-03-12. 4.3

[OMG04b] Object Management Group, Inc. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, April 2004. ad/2002-04-10. 2.4.2

[OMG05a] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, November 2005. Final Adopted Specification, ptc/05-11-01. (document), 2.4.1, 2.17

[OMG05b] Object Management Group, Inc. *MOF 2.0/XMI Mapping Specification*, September 2005. Version 2.1, formal/05-09-01. 2.3.2

[OMG05c] Object Management Group, Inc. *Unified Modeling Language: Superstructure*, August 2005. Version 2.0, formal/05-07-04. (document), 1.1.1, 2.2, 2.8, 2.3.3, 2.12

[OMG06a] Object Management Group, Inc. *CORBA Component Model Specification*, April 2006. Version 4.0, formal/06-04-01. 2.2.4

[OMG06b] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Version 2.0, OMG Available Specification, formal/06-01-01. (document), 2.3, 2.3.1, 2.7

[OMG06c] Object Management Group, Inc. *Ontology Definition Metamodel*, 2006. Sixth Revised Submission to OMG/ RFP ad/2003-03-40, ad/2006-05-01. 4.8, A

[OMG06d] Object Management Group, Inc. *Unified Modeling Language: Infrastructure*, March 2006. Version 2.0, formal/05-07-05. 2.3.4

[PVW+04] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In Panos Markopoulos, Berry Eggen, Emile H. L. Aarts, and James L. Crowley, editors, *Proceedings of the Second European Symposium on Ambient Intelligence (EUSAI 2004), Eindhoven, The Netherlands*, volume 3295 of *Lecture Notes in Computer Science*, pages 148–159. Springer-Verlag, November 2004. 1.4, 4.3, 8.3.1

[RB06] Stephan Roser and Bernhard Bauer. An approach to automatically generated model transformation using ontology engineering space. In *Proceedings of the 2nd Workshop on Semantic Web Enabled Software Engineering, Athens, GA, USA*, November 2006. 4.8

[RDH+04] Alan Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In Enrico Motta, Nigel Shadbolt, Arthur Stutt, and Nick Gibbins, editors, *Proceedings of the European Conference on Knowledge Acquistion, Northampton, England*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag, September 2004. 3.4

[RMR05] S. Reiff-Marganiec and M. D. Ryan, editors. *Proceeding of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05), Leicester, UK*. IOS Press, June 2005. 5.3.1, 6.2, 6.6

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007. 4.2, 7.1

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. *OWL Web Ontology Language Guide*. World Wide Web Consortium, February 2004. W3C Recommendation 10 February 2004, [Online] http://www.w3.org/TR/owl-guide/. 1.3.1, 2.5, 3.1, 3.2, 3.3

[SZW05] Jing Sun, Hongyu Zhang, and Hai Wang. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05), Shanghai, China*, pages 303–312. IEEE Computer Society, June 2005. 5.3.1

[TBA04] Bedir Tekinerdoğan, Sevcan Bilir, and Cem Abatlevi. Integrating platform selection rules in the model driven architecture approach. In Uwe Aßmann, Mehmet Akşit, and Arend Rensink, editors, *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Enschede, The Netherlands, June 2003 and Linköping, Sweden, June 2004. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, August 2004. 2.1, 4.8

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006)*, number 4130 in Lecture Notes in Artificial Intelligence, pages 292–297. Springer-Verlag, October 2006. 4.2, 7.1

[Tob01] Stephan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, Aachen, Germany, May 2001. 4.7.2, 8.4.2

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, CA, USA*, pages 92–93. ACM Press, October 2003. 1.3.2, 5.4

[VB01] Ragnhild Van Der Straeten and Johan Brichau. Features and feature interaction in software engineering using logic. In Elke Pulvermüller, Andreas Speck, James Coplien, Maja D'Hondt, and Wolfgang De Meuter, editors, *Proceedings of the ECOOP 2001 Workshop on Feature interaction in composed systems, Budapest, Hungary*, volume 2001 of *Interner Bericht*, pages 79–88. Universität Karlsruhe, June 2001. 5.3.1

[Wag05] Dennis Wagelaar. Context-driven model refinement. In Uwe Aßmann, Mehmet Akşit, and Arend Rensink, editors, *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Enschede, The Netherlands, June 2003 and Linköping, Sweden, June 2004. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, August 2005. 1.4, 8.3.1, 8.3.2, 8.3.4, 8.3.5

[Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *Accepted for the International Conference on Model Transformation (ICMT 2008), Zürich, Switzerland*, Lecture Notes in Computer Science. Springer-Verlag, July 2008. 1.4, 2.4.2, 8.3.6, 8.5.3

[WF94] Christopher A. Welty and David A. Ferrucci. What's in an instance? Technical Report 94-18, RPI Computer Science Dept., Troy, NY, USA, 1994. 2.3.4

[WF99] Christopher A. Welty and David A. Ferrucci. Instances and classes in software engineering. *Intelligence*, 10(2):24–28, 1999. 3.3

[WJ05] Dennis Wagelaar and Viviane Jonckers. Explicit platform models for MDA. In *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica*, volume 3713 of *Lecture Notes in Computer Science*, pages 367–381. Springer-Verlag, October 2005. 1.4, 8.3.1, 8.3.2, 8.3.4, 8.3.5

[WLS+07] Hai H. Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. Verifying feature models using OWL. *Journal of Web Semantics*, 5(2):117–129, June 2007. 5.3.1, 6.6

[WSWN07] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC 2007), Kyoto, Japan*, pages 129–140. IEEE Computer Society, September 2007. 6.6

[WV06] Dennis Wagelaar and Ragnhild Van Der Straeten. A comparison of configuration techniques for model transformations. In Arend Rensink and Jos Warmer, editors, *Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain*, volume 4066 of *Lecture Notes in Computer Science*, pages 331–345. Springer-Verlag, July 2006. 1.4, 8.3.3

[WV07] Dennis Wagelaar and Ragnhild Van Der Straeten. Platform ontologies for the model-driven architecture. *European Journal of Information Systems*, 16(4):362–373, August 2007. 1.4, 8.3.1, 8.3.2, 8.3.3, 8.3.4, 8.3.5, 8.3.6

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004), Seattle, WA, USA*, volume 3308 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, November 2004. 5.3.1

Index