Model-Driven Evolution of Software Architectures

Model-Driven Evolution of Software Architectures

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op 27 november 2007 om 12:30 uur

door

Bastiaan Stephan GRAAF

informatica ingenieur
geboren te Den Haag

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. A. van Deursen

Samenstelling promotiecommissie:

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

The research described in this thesis was made possible through suppport of SenterNovem (Moose project) and NWO (Reconstructor project).

IPA Dissertation Series 2007-16

About the cover: The cover shows a fragment of Johannes Vermeer: 'View on Delft', Mauritshuis, The Hague (ca. 1661-1664. Oil on canvas) and a picture taken from the same viewpoint by René van der Krogt in May 2006.

Printed by Universal Press, Veenendaal

ISBN: 978-90-9022390-2

Preface

At last! Time has come to write the preface to my PhD-thesis. During the almost five and a half years it took me to arrive at this point, I started to compare the process of doing a PhD with the running exercises I do in between two football seasons. I always start very enthusiastic and motivated; halfway I doubt that I will ever make it and almost regret to have started; near the end I try hard to put in all remaining energy to make it to the finish; and afterwards I feel good, happy, and proud about what I've done. Like I feel now. One big difference is that during my PhD-career I encountered many people that helped me or that made the whole exercise more pleasant. Those people deserve to be acknowledged. Well, here it goes.

If crossing the finish line has to be attributed to one person, than that person has got to be Arie van Deursen. Arie, I am very grateful to you for 'adopting' me as a PhD-student. For me the best remedy to overcome seemingly dead ends in my research or when writing articles was a conversation with you. You have been a truly inspiring teacher! For this and many other reasons it was a pleasure to work with you.

I want to thank Hans Toetenel and Jan Dietz for talking me into this excercise in the first place. I am also grateful for Hans' supervision during the first phase of my work as a PhD-student.

Of course somebody has to read the result of all that work. For this, my gratitude goes to the members of the examination committee: Prof. H.J. Sips, Prof. A. Verbraeck, Prof. M.G.J. van den Brand, Prof. R. Koscke, and Dr. L. Somers.

I've had the opportunity to work in several research projects: Moose, Merlin, Ideals and Reconstructor. During all these projects I worked together with interesting and inspiring people. Thank you all for that! More in particular, I want to thank Rini van Solingen for all kinds of general advice on how to succeed as a PhD-student. Also I want to specially thank Sven Weber as a co-author of one of the publications on which this thesis is based.

Furthermore, I want to mention all (former) colleagues of the software engineering department for all the nice lunches, drinks, and diners. In particular I am grateful to Hylke van Dijk and Marco Lormans. Hylke as a co-author of two articles on which this thesis is based, and for asking those nasty questions all the time. With Marco I worked together already since the beginning of our master-thesis project seven years ago, all that time as roommates. Most working trips have been quite memorable because of him. Thanks!

Finally, I want to thank Daan for being patient enough to cope with me being *almost finished* for a such a long time and, putting on hold all other plans; for being impatient enough to apply the right amount of pressure; and for being my girl-friend all the way.

Bas Graaf
Delft, August 2007

Contents

Preface v

List of Figures x

Abbreviations xiii

1 Introduction 1

1.1 Problem Description: Evolution of Software Architectures . 4

1.2 Objectives 7

1.2.1 Integration in Practice 7

1.2.2 Product Lines 7

1.2.3 Model-Driven Engineering 8

1.3 Approach 9

1.4 Overview 10

1.5 Origin of Chapters 13

2 Background **17**

2.1 Software Evolution 17

2.2 Architecture-Driven Software Development 20

2.2.1 Software Architecture 20

2.2.2 Software Architecture Usage 22

2.2.3 Software Architecture Design 23

2.2.4 Software Architecture Description 24

2.2.5 Software Architecture Evaluation 25

2.2.6 Software Product Lines 26

2.3 Model-Driven Engineering 27

2.3.1 Modelling 28

2.3.2 Metamodelling 29

2.3.3 Model Transformations 32

2.3.4 MDE and Other Technological Spaces 37

2.4 Model-Driven Evolution of Software Architectures 38

3 Embedded-Software Engineering: The State of the Practice 41
3.1 Introduction 41
3.2 Methods and Scope 42
3.3 Embedded-Software Development Context 45
3.4 Requirements Engineering Results 47
3.4.1 Requirements Specification 48
3.4.2 Requirements Management 49
3.5 Software Architecture Results 50
3.5.1 Software Architecture Design 50
3.5.2 Software Architecture Description 51
3.5.3 Software Architecture Evaluation 52
3.5.4 Reuse 53
3.6 Discussion 54
3.7 Outlook 55

4 Evaluating an Embedded Software Reference Architecture – Industrial Experience Report – 57
4.1 Introduction 57
4.2 Overview of the Reference Architecture 59
4.2.1 Business Drivers 59
4.2.2 Reference Architecture 60
4.2.3 Structure 61
4.2.4 Usage 61
4.3 Evaluation Approach 63
4.3.1 Selection of Evaluation Method 63
4.3.2 Tailoring SAAM 65
4.4 Conducting the Evaluation 67
4.4.1 Preparation 67
4.4.2 Scenarios 67
4.4.3 Execution 68
4.4.4 Overall Evaluation 70
4.5 Discussion 71
4.5.1 Reference Architecture 71
4.5.2 Distributed SAAM 73
4.6 Related Work 75
4.7 Conclusion 76

5 Model-Driven Consistency Checking of Behavioural Specifications 77
5.1 Introduction 77
5.2 Running Example 80

5.3 Related Work 82
5.4 Model-Driven Consistency Checking 83
5.4.1 Enabling Technologies 83
5.4.2 Behavioural Modelling 85
5.4.3 Consistency Checking Approach 87
5.5 Generating State Machines 88
5.5.1 Instantiating a Source Model 88
5.5.2 Model Transformations 89
5.6 Application to Océ 94
5.6.1 Source Model Normalisation 94
5.6.2 Results 95
5.7 Discussion 98
5.8 Conclusions 101

6 Model-Driven Conformance Checking of Structural Specifications 103
6.1 Introduction 103
6.2 Running Example 105
6.3 Approach 108
6.3.1 Conformance Checking System 108
6.3.2 Model-Driven Conformance Checking 109
6.4 Viewpoints and Metamodels 110
6.4.1 Component-and-Connector Views 111
6.4.2 Module Views 112
6.5 Mappings and Model Transformations 114
6.5.1 Model Population 114
6.5.2 Model Comparison 118
6.5.3 Model Presentation 121
6.6 Discussion 121
6.6.1 Modelware 121
6.6.2 Improving the Approach 123
6.7 Related work 125
6.8 Conclusions 126

7 Model-driven Migration of Supervisory Machine Control Architectures 129
7.1 Introduction 129
7.2 Related Work 132
7.3 Migration Context 133
7.3.1 Supervisory Machine Control 133
7.3.2 Running Example: A Wafer Scanner 134
7.3.3 Concerns for Supervisory Machine Control Systems . 135
7.4 Model-Driven Migration 136

7.5 Source Metamodel . . . 138
7.6 Normalisation Rules . . . 143
7.7 Target Metamodel . . . 147
7.8 Transformation . . . 151
7.8.1 The Atlas Transformation Language . . . 152
7.8.2 Basic Target Model Elements . . . 153
7.8.3 Concern-Based Transformation Rules . . . 156
7.8.4 Transformation Results . . . 159
7.9 Evaluation . . . 161
7.10 Conclusions and Future Work . . . 165

8 Visualisation of Domain-Specific Modelling Languages Using UML 167
8.1 Introduction . . . 167
8.2 Background . . . 169
8.2.1 Software Architecture . . . 169
8.2.2 Enabling MDE Technologies . . . 171
8.3 Model-Driven Architectural Views . . . 172
8.3.1 MDAV Framework . . . 172
8.3.2 MDAV Process . . . 173
8.4 Using MDAV to Generate Views . . . 174
8.4.1 Module-Uses View . . . 174
8.4.2 Component-and-Connector View . . . 177
8.5 Industrial Application . . . 181
8.6 Discussion . . . 185
8.7 Related Work . . . 186
8.8 Concluding Remarks . . . 187

9 Conclusion 189
9.1 Contributions . . . 190
9.2 Integration in Practice (*RQ1*) . . . 190
9.3 Software Product Lines (*RQ2*) . . . 193
9.4 Model-Driven Engineering (*RQ3*) . . . 194
9.5 Support for Evolution of Software Architectures (*RQ0*) . . . 199
9.6 Future Work and Recommendations . . . 201

References 205

Summary 221

Samenvatting 225

Curriculum Vitae 229

List of Figures

1.1 ASML wafer scanner 2
1.2 Software evolution tasks 5
1.3 Océ copier 12

2.1 Complexity vs. size for subsequent revisions of copier software 18
2.2 Fundamental relations between system, model, and metamodel 28
2.3 Layered MDE modelling stack 30
2.4 Metamodel and conforming model 31
2.5 Conforms to relation in MDE 31
2.6 Model transformation megamodel 33
2.7 A metamodel for simple class models 34
2.8 Three-dimensional evolution framework 38

3.1 The decomposition of the embedded-systems-development process 46

3.2 Embedded-systems-development stakeholders and other factors 47

4.1 Main flows in a copier. 59
4.2 The reference architecture and derived projects. 62
4.3 SAAM steps 64
4.4 SAAM results 69

5.1 Typical development process 78
5.2 Architecture for copier engines 81
5.3 Constraints 85
5.4 Collaborations (simplified) 86
5.5 State machines 86
5.6 Flat state machine 92

5.7 Example scenario: request a copier engine to go to standby while it is running . . . 96
5.8 Merged state model of ESM (fragment) . . . 97

6.1 Aligning architecture and implementation . . . 104
6.2 Digital music box reader system . . . 106
6.3 Architectural views . . . 107
6.4 Conceptual Conformance Checking System . . . 109
6.5 Generic metamodel element . . . 111
6.6 Metamodels . . . 113
6.7 Reconstructed MADL models . . . 117
6.8 Reconstructed CPADL models . . . 119
6.9 Merged MADL conformance model . . . 119
6.10 Merged CPADL conformance model . . . 120

7.1 Machine control context . . . 133
7.2 Simplified layout of a wafer scanner . . . 134
7.3 Generic two-phased migration approach . . . 137
7.4 Source metamodel . . . 139
7.5 Process wafer request . . . 141
7.6 Unload wafer request . . . 142
7.7 Normalised process wafer request . . . 148
7.8 Normalised unload wafer request . . . 149
7.9 Module view for the product-line SMC architecture . . . 150
7.10 Target metamodel . . . 151
7.11 Results for unload wafer request . . . 160
7.12 One of three concurrent state models (made anonymous) . . 164

8.1 MDAV framework . . . 172
8.2 C&C model of CaPiTaLiZe (ACME) . . . 174
8.3 MADL . . . 175
8.4 CCADL . . . 178
8.5 XML metamodel . . . 180
8.6 Task-resource metamodel, model, and UML representation . 184

9.1 Megamodel for model-driven evolution of software architectures . . . 195

Abbreviations

ADL	architecture description language
ALMA	architecture-level modifiability analysis [Bengtsson et al., 2004]
API	application programming interface
AST	abstract syntax tree
ATAM	Architecture Tradeoff Analysis Method [Clements et al., 2002b]
ATL	Atlas Transformation Language [Jouault and Kurtev, 2005]
COTS	commercial off-the-shelf
CMM	Capability Maturity Model [Humphrey, 1989]
DSAAM	Distributed SAAM
DSL	domain-specific language
DSML	domain-specific modelling language
DTD	Document Type Definition
EBNF	Extended Backus-Naur form
EMF	Eclipse Modeling Framework[1]
FSM	finite state machine
GMF	Graphical Editing Framework[2]
GPL	general-purpose language
ITEA	Information Technology for European Advancement[3]

[1]http://www.eclipse.org/emf (June 2007)
[2]http://www.eclipse.org/gmf (June 2007)
[3]http://www.itea-office.org (June 2007)

MDA	Model Driven Architecture[1]
MDE	model-driven engineering
MDR	Metadata Repository[2]
MOF	MetaObject Facility[3]
MOOSE	Software Engineering *M*eth*O*d*O*logie*S* for *E*mbedded Systems[4]
OCL	Object Constraint Language[5]
OMG	Object Management Group[6]
QVT	Query/View/Transformation [OMG, 2005]
SAAM	Software Architecture Analysis Method [Kazman et al., 1996]
SEI	Software Engineering Institute[7]
SMC	supervisory machine control
SPICE	Software Process Improvement and Capability dEtermination [Emam et al., 1997]
SVG	Scalable Vector Graphics[8]
UML	Unified Modeling Language[9]
XMI	XML Metadata Interchange[10]
XML	Extensible Markup Language[11]
XSLT	Extensible Stylesheet Language Transformations[12]

[1]http://www.omg.org/mda (June 2007)
[2]http://mdr.netbeans.org (June 2007)
[3]http://www.omg.org/mof (June 2007)
[4]http://www.mooseproject.org (June 2007)
[5]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)
[6]http://www.omg.org (June 2007)
[7]http://www.sei.cmu.edu (June 2007)
[8]http://www.w3.org/Graphics/SVG (June 2007)
[9]http://www.uml.org (June 2007)
[10]http://www.omg.org/mda/specs.htm#XMI (June 2007)
[11]http://www.w3.org/XML (June 2007)
[12]http://www.w3.org/TR/xslt (June 2007)

Chapter 1

Introduction[1]

Most software that is really used is exposed to many forces that require it to change, such as changing user requirements or a changing operating environment. As a result software changes continuously. This process is called software evolution [Lehman and Belady, 1985].

When changes are required to a software system, the question is whether they can be implemented within the bounds set by the current architecture or require a redesign of the architecture.

The former causes types of software evolution referred to as architectural drift and erosion [Perry and Wolf, 1992]. Architectural drift occurs when the current architecture is not well-understood by the developers involved in making these small-scale changes. As a result their changes are based on a software architecture that is different from the intended architecture. Architectural erosion is caused by violations of the architecture. Both have a negative effect on the maintainability of software.

Eventually architectural drift and erosion make a redesign of the architecture unavoidable. In this thesis we consider this type of software evolution, that is, on the level of architecture. Although we discuss the concepts of software architecture extensively in Chapter 2, for now it suffices to describe architecture as the high-level or global design of a software system.

To illustrate the need for and implications of architectural changes, consider the following situation at ASML, a company that develops manufacturing machines for the semiconductor industry. A new architecture for the control software of one of ASML's products, a so-called wafer scanner (see Figure 1.1 on the following page), is investigated [Van den Nieuwelaar et al., 2003]. This architecture is based on generic reusable software components and enables the (automatic) generation of application-specific parts

[1]This chapter is based on: Graaf, Bas. Model-driven evolution of software architectures. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 357–360. IEEE Computer Society, 2007

Figure 1.1: ASML wafer scanner

of control components from a declarative specification. As a result, the required effort for development and maintenance of the control software can be reduced.

A problem that remains is the migration of existing control components to this new architecture. A possible approach is to start-over and develop these components from scratch according to the new architecture. In this case this means that for each control component a new specification has to be created. Unfortunately, the knowledge incorporated in the code and designs of already existing control components will be largely lost this way. Considering the fact that the current behaviour of these components does not need to be changed, this constitutes a waste of knowledge and resources, and an unnecessary risk. An alternative solution is to derive the specifications of these components for the new architecture from their existing specifications.

The scenario sketched above, which is fully explored in Chapter 7, is about a changing or evolving software architecture and illustrates some of the issues we address in this thesis. Many companies are confronted with similar scenarios. An example in a completely different domain is the problem of making existing information systems accessible via the World Wide Web. This typically also requires architectural changes. Many other scenarios can be mentioned that involve architectural changes for reasons that include maintainability and functionality.

However, while often required, architectural changes typically come with a significant risk and are expensive to perform. Moreover, when the objective of such changes is maintainability improvement, as in the scenario above, their benefits are only experienced later on. This makes such changes problematic. Therefore, our goal is to support evolution of software architectures such that risk and costs are reduced.

Considering the described scenario, next to the question of how to migrate existing components, other questions arise, such as how to evaluate that such changes are necessary, and, whether they can be performed (semi-) automatically.

Another relevant aspect of this scenario is that it deals with a software architecture based on a platform (i.e., the generic reusable components), which applies to a whole series of systems (i.e., the different control systems in a wafer scanner). Such architectures are referred to as product-line architectures [Clements and Northrop, 2002]. To realise economies of scale, a trend in industry is to integrate the development of a whole set of similar products in a single (software) product line. The development of a software product line is based on a product-line architecture that defines the commonality and variability between the product-line members (i.e., individual products of the product line). In the context of software architecture evolution, product-line architectures are a complicating factor and, as we will explain in this thesis, require special attention.

Another complicating factor to achieve our goal of supporting software architecture evolution is the integration of evolution support in industrial practice. Industry is reluctant to adopt new software engineering technologies. An important reason for this is that it tends to have a risk-avoiding attitude. This problem is also addressed in this thesis.

We will search the solution in the area of *model-driven engineering*. We adopt the vision of model-driven engineering (MDE) for the purpose of supporting software evolution. MDE is the term for a new generation of software development approaches in which models play a dominant role and in which (part of) the development steps are performed by (automatic) model transformations [Bézivin, 2005]. In these approaches software models are gradually transformed into source code, which typically executes on top of a software platform.

MDE approaches are enabled by the availability of standards, such as for modelling and transformations. They have been developed to hide the behavioural and structural complexity of the platforms underlying software product lines [Schmidt, 2006]. This corresponds to the envisioned situation in the scenario we described above. The concept of MDE is further discussed in Chapter 2.

Our approach is to investigate to what extent software architecture can be made explicit as models, and whether the existing knowledge, standards, and tools in the area of MDE can be used for the purpose of software architecture evolution. Thus, instead of applying model transformations for the *development* of software by the generation of source code from more abstract software models, we apply model transformations to support the *evolution* of software. This involves different types of software engineering tasks, such as evaluation and migration. We investigate the extent to which such tasks can be performed by the use of model transformations. Additionally, we focus on real-life situations such as the migration of control components at ASML.

In the remainder of this introduction we describe the problem and formulate the research questions this thesis addresses. Subsequently, we explain the approach we followed to answer these questions and the scope of our work. We conclude with an outline of this thesis and an overview of its contributions.

1.1 Problem Description: Evolution of Software Architectures

In this thesis, we focus on the evolution of software platforms and the systems they support. More particularly, we address the problem of their evolution on the architectural level.

Perry and Wolf [1992] describe architecture as the 'load-bearing walls' of a software system. As such, a software architecture allows some changes and precludes others, that is, it allows some degree of evolution. Changes that it allows do not require a migration of the architecture. Changes, however, that are not supported by the current architecture will require such a migration. As such, an architecture determines which type of evolution is cheap (i.e., the type that involves changes that do not require changes to the architecture) and which type is expensive (i.e., the type that involves changes that do require changes to the architecture). In fact, a reason for migrating to a different software architecture is to change this, that is, making a different type of changes cheap. As an example, in the ASML scenario sketched above one of the goals was indeed to reduce the effort required to change the sequence of the manufacturing activities a wafer scanner performs for the manufacturing of microchips.

When considering software evolution from an architectural perspective, it needs to be determined *if* an architecture requires changes, and subsequently *how* to perform those changes. The former requires an architecture evaluation. The latter requires an approach to migrate a software architecture, and the corresponding 'downstream' development artefacts. In the case of a complex architecture, or a product line, where an architecture affects multiple systems, it pays off to do this automatically. To do so, an architecture can be considered as a model that can be manipulated. The technology to make this happen is offered by MDE. In-line with MDE, we aim at the development of automated techniques, where possible.

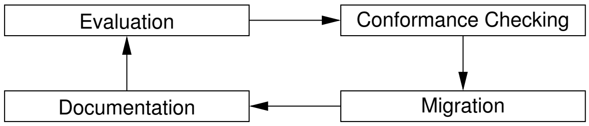


Figure 1.2: Software evolution tasks

As we will see from this thesis, the automatic manipulation of (architectural) models is hampered by industry's resistance to adoption of state-of-the-art software engineering technologies. An important reason for this is that such technologies often have a large impact on current ways of working, resulting in unacceptable risks (see also Chapter 3). This means that, in the context of software evolution, we have to take into account, for instance, the informal use of modelling languages in industry [Lange et al., 2006]. This makes automation particularly difficult. In general, the impact of solutions (technologies or processes) to current ways of working should be minimised.

To clarify the scope of our work we distinguish four types of activities related to evolution of software. We refer to these activities as software evolution tasks. The tasks we consider are depicted in Figure 1.2 in an evolutionary software life-cycle and are explained below.

Evaluation In our work the main objective of architecture evaluation is to determine whether or not proposed changes to a software system require changes to the current architecture. We consider architecture evaluation as the starting point of a software architecture evolution cycle. A particular challenge is the assessment of whether a *product-line* architecture requires changes in the face of anticipated changes to the product-line members. Dobrica and Niemelä [2002] give an overview of proposed architecture evaluation approaches. However, none of those is explicitly aimed at software product lines.

Conformance Checking In the case that an evaluation indicates that architectural changes are required, it is necessary to determine to what extent 'downstream' development artefacts conform to the (product-line) architecture. The question is whether development artefacts that are constrained by the decisions made during the architecting phase do not violate these decisions. In principle, all design artefact are constrained by the architecture, such as detailed designs, implementations and even product-line members.

Krikhaar [1999] and Mens [2000] compare a number of approaches to check architecture conformance. However, conformance between models at different abstraction levels is not addressed. Moreover, most approaches dictate the introduction of specific modelling languages, requiring a change to current ways of working.

Migration A set of consistent development artefacts as determined by the conformance checking task, reduces the risk of an actual migration of the architecture and dependent development artefacts. The migration to a new product-line architecture and associated software platform that better supports foreseen requirements, requires the migration of all products supported by the legacy platform. There is no previous work that considers software (architecture) migration as a model transformation problem. Several other work does address the transformation of software systems. However, they consider single-product architectures [Bosch and Molin, 1999], simple graphs [Fahmy and Holt, 2000b], or the level of source code [Terekhov and Verhoef, 2000]. The language migration process used by Terekhov and Verhoef [2000] is particularly interesting. It separates a migration in three phases that include restructuring of source programs to enable the (automatic) transformation phase. Although it was used for source code migration, such a preparatory step is also required for the migration on the architectural level to take into account industrial modelling conventions.

Documentation After a migration of the (product-line) architecture and the product-line members it supports, documentation needs to be updated. It is generally accepted that the documentation of software architectures consists of multiple views [Kruchten, 1995; Hofmeister et al., 2005]. Often the Unified Modeling Language[1] (UML) is used in these views. On the other hand, specialised architecture description languages (ADLs) (see Medvidovic and Taylor [1997] for an overview) and MDE support the creation of models to automate several software engineering tasks, such as code generation. However, no approach addresses the problem of keeping documentation and models consistent. With the upcoming of MDE approaches this becomes a highly relevant problem.

In this thesis, we aim at increasing our understanding of each of these four software evolution tasks as well as offering support for them.

[1]http://www.uml.org (June 2007)

1.2 Objectives

In the previous section we explained that changes to software architectures can be required to improve or restore the maintainability of software systems. However, such evolution involves high risks and costs.

As such, our main research question is:

RQ0 How can the evolution of software architectures be supported?

We will investigate this question in terms of the software evolution tasks we identified: evaluation, conformance checking, migration, and documentation. When considering the problem description in Section 1.1, *RQ0* raises a number of subquestions that we introduce below.

1.2.1 Integration in Practice

As we will see in this thesis, integration of new software engineering technologies in industrial practice is difficult. This is due to the risk-avoiding attitude of industrial companies towards such innovation, resulting in a preference of proven technologies.

However, also academia's attitude towards practical industrial problems hampers application of research results in practice. These problems are often considered not to be interesting from an academic point of view or are difficult to investigate because such investigations are very costly and time consuming. Finally, industry is not always willing to cooperate. The result is that often software engineering technologies developed by academia are not (fully) applied in industrial practice. An example is the informal use of modelling in practice.

This leads to our first subquestion:

RQ1 How to integrate the support for software evolution tasks in practice, considering the informal use of modelling languages and preference for proven technologies in industry?

1.2.2 Product Lines

Many companies extended the scope of their software architectures from single systems to multiple systems to increase reuse and reduce required development and maintenance effort[1].

For our software evolution tasks the use of product line principles is relevant. One reason is, for instance, that product line architectures are

[1]For examples, visit the Product Line Hall of Fame: http://www.sei.cmu.edu/productlines/plp_hof.html (June 2007)

defined on a higher level of abstraction than single-product architectures. Furthermore, the number of stakeholders for a product line also is higher. This complicates, for instance, evaluations.

This leads to our second subquestion:

RQ2 What is the impact of the use of software product lines and platforms on the support for software evolution tasks?

1.2.3 Model-Driven Engineering

Software architecture evolution is costly and risky. Therefore, we will investigate the use of MDE technology for this problem. Automation is one of the key characteristics of MDE. When applied to the architecture evolution this may yield cheaper and more reliable results.

Our use of MDE is also motivated by *RQ1* and *RQ2*. The development of MDE technologies has been driven by industry. This can be seen, for instance, from the wide-spread use of UML for software design. As such, support for software evolution tasks based on similar technology might by itself already improve integration of such support in industrial environments.

Finally, a strong link exists between MDE and software product lines. With MDE the generated code typically executes on top of a software platform. At the same time software platforms are the foundation for even the most basic product lines [Bosch, 2002]. As such, MDE approaches can be used for the automatic derivation of product-line members [Deelstra et al., 2003].

For industrial applicability, one specific type of MDE is particular relevant. We focus on MDE technologies based on a set of standards defined by the Object Management Group[1] (OMG) under the name Model Driven Architecture[2] (MDA). The reason for this is that UML is an essential part of the MDA framework; and UML is the (de facto) standard for modelling software [Kobryn, 1999] that is most widely applied in industry. We believe that the practical relevance of our work is increased by restricting ourselves to this framework (see also *RQ1*).

Model-driven support at the architectural level for our software evolution tasks allows for (partial) automation, resulting in improved reliability, efficiency (of the development process), and quality (of developed software) [Atkinson and Küne, 2003; Selic, 2003].

This leads to our third and final subquestion:

RQ3 To what extent can the support for software evolution tasks be automated by the use of model-driven engineering?

[1]http://www.omg.org (June 2007)
[2]http://www.omg.org/mda (June 2007)

1.3 Approach

As software engineering is an applied science, our view on software engineering research is that results can only be proven useful by validation in industrial practice. Furthermore, this type of research is aimed at solving real problems. Such problems are mainly found in industry (at least problems in the domain of software engineering).

Therefore, we intend our research to be industry-driven; we adopt the 'industry-as-laboratory' approach proposed by Potts [1993]. In this approach the problems studied are identified by close involvement in industrial projects and results are applied to practical problems; there is an emphasis on real case studies.

We accomplish the interactions with industry on which this approach is based in three ways: a survey, industrial case studies, and close collaboration with software practitioners in industry.

We first perform a survey among more than 35 software practitioners at eight companies to get an overview of software engineering practices and specific problems in the (embedded) software industry (see Chapter 3). The observations made in that survey include the upcoming use of product-line approaches, the informal use of modelling languages, and the importance of the evolutionary aspect of software. This survey partially determined the problems we address in the research described by this thesis.

The exploratory character of our research, the type of research questions we want to investigate ('how' questions), and the low level of control we have over the (industrial) environment in which software evolution takes place, make that case studies are a suitable research approach [Yin, 2003]. Furthermore, the use of industrial case studies reduces the risk of scalability problems of the results Kitchenham et al. [1995]. Therefore, we use case studies to investigate the applicability of model-driven approaches to the four software evolution tasks we defined.

For each task we propose a separate solution, which we evaluate in a (industrial) case study. As such, we performed separate case studies for each of the evolution tasks: evaluation, conformance checking, migration, and documentation. The case studies we conducted are mainly related to two industrial systems: copiers developed by Océ and wafer scanners developed by ASML. As such, by our case studies, we focus on the embedded software domain. The conclusion in Chapter 9 also reflects on the question of whether this is relevant from the perspective of our research questions.

Given the exploratory character of our research, we do not define a set of research propositions to investigate. Instead, we direct our research by the subquestions outlined in Section 1.2. As discussed by Yin [2003] we use this direction to guide our analysis of the case studies we perform. We qualitatively evaluate the solutions we propose by carefully observing and analysing their application in each case study.

For the case study in which we evaluated our approach for the migration task, we were able to compare our findings with respect to a migration of the same system conducted manually. The conformance checking tasks were only executed using our techniques. Therefore, their evaluation is based on the type and number of inconsistencies found. For the evaluation and documentation tasks, we evaluate our solutions with respect to the application of similar approaches in other cases.

Considering our research questions, we specifically focused the evaluation on the extent to which the software evolution tasks can be automated, the impact of software product lines, and possibilities for reusing (proven) software technologies and reducing organisational impact.

1.4 Overview

This thesis addresses the problem of managing evolution for complex software intensive systems. We studied this problem and its solutions in terms of software architecture and MDE. The remainder of this thesis is organised as follows.

Setting the Scene In Chapter 2 we introduce some of the concepts that were touched upon only briefly in this introduction more thoroughly. In particular we discuss software architecture and MDE.

Chapter 3 reports on the survey we conducted among several companies developing embedded software. The survey resulted in a number of important observations for this thesis:

- Industry rarely develops products from scratch. This observation confirms the importance of the evolution and maintenance aspects of software development.
- Increasingly, product-line and MDE approaches are applied for the development of embedded software.
- Current software engineering technologies are difficult to apply in practice due to several reasons. One consequence is that such technologies are applied in a pragmatic way, for instance, modelling languages and tools are often only used to *draw* diagrams for the purpose of documentation rather than for the purpose of, for instance, automatic analyses or code generation.

The first two observations call for an approach that enables the introduction of product lines in a "bottom-up" manner, meaning that product-lines come into existence based on existing products and are developed in an evolutionary, rather than revolutionary (or top-down), way.

The third observation adds the constraint that such an approach takes into account some of the practical issues that are a reality for software development organisations, such as the informal use of modelling languages, the limited amount of time for doing analysis, and the risk involved in changing existing ways of working.

The Evaluation Task In Chapter 4 we define a scenario-based approach based on the Software Architecture Analysis Method (SAAM) [Kazman et al., 1996] for assessing the quality of an emerging product-line architecture for the embedded software for copiers developed by Océ (Figure 1.3 on the following page). This architecture emerged in a bottom-up manner from a number of existing products. At some point questions were raised with respect to the suitability of the product-line to incorporate more existing and future products as product-line members. Therefore an assessment was initiated that had to take into account the emergent character and corresponding low-visibility of the product-line in the organisation. The latter resulted in a low commitment of several stakeholders to such an assessment. The results show that a two-phased scenario development step, in which part of the scenarios are collected separately from the joint evaluation session, results in a more efficient approach that still yields acceptable assessment results. Additionally, this chapter identifies the problem of conformance of the architecture of product-line members to a product-line architecture. Such conformance is desirable before updating a product-line architecture or migrating an existing product to incorporate it in the product-line.

The Conformance Checking Task Two chapters deal specifically with conformance checking.

In Chapter 5 we discuss how to use model transformations to combine scenarios into state-based behavioural models. Compared to the previous case study, the scenarios are expressed in more detail using UML sequence diagrams. We applied the transformations to a set of scenarios for a component defined by the product-line architecture for the embedded copier software of Océ. This results in a state transition model. To assess the extent to which the scenarios are consistent with a state transition model that is used to generate the source code for that component, we (manually) investigate the differences between the two state models. As such, we identified a number of inconsistencies.

Figure 1.3: Océ copier

A model-driven approach for automatically determining the conformance of software artefacts is proposed in Chapter 6. A view-based process for conformance checking is described that does not interfere with the current way of working of the involved domains (e.g., requirements, architecture, or implementation) by introducing the concept of a conformance viewpoint (i.e., a type of view). In the case of architecture conformance such a viewpoint, specified as a metamodel, defines checkable aspects of the architecture and implementation, as such bridging the semantic gap between the two domains. We illustrate how model transformations can be used to automatically discover inconsistencies between architecture specifications and implementation.

The Migration Task Once a product-line architecture has been assessed and the conformance of product-instances with respect to the product-line has been confirmed, existing products need to be migrated to the new product-line approach. This requires that instance specific information is extracted and transformed into a view associated with a viewpoint that was defined to describe product instances. In Chapter 7 we describe a generic view-based process for migrating the legacy designs discussed at the start of this chapter into views that exactly describe a product instance in terms of a new product-line architecture. For the migration of control architectures based on finite state machines we define a number of transformation rules that result in a specification of such an architecture in terms of a product-line architecture based on task-resource systems. These rules are amenable for an MDA-type of approach that is partially automated.

Table 1.1: Coverage of research questions in chapters

Question \ Chapter	3	4	5	6	7	8
RQ1	√	√	√	√	√	√
RQ2		√			√	
RQ3			√	√	√	√

The Documentation Task To decrease the effort required for future evolution of software products, up-to-date documentation is an important asset [Forward and Lethbridge, 2001]. In Chapter 8 we discuss the relation between the architectural models used for conformance checking and migration, for instance, and architectural views for documentation. We present a framework in which the involved concepts are related to each other and show how such a framework can be supported by MDE technologies.

The Research Questions Finally, to conclude, Chapter 9 presents an overview of our contributions and revisits the research questions raised in Section 1.2. Table 1.1 illustrates how these questions are covered by the core chapters of this thesis. All chapters take into account the practical applicability of the proposed solutions, for instance by the use of two industrial case studies. Development using product-line principles plays an important role in Chapters 4 and 7. Automation using MDE is the dominant concern in the final four core chapters. In all these cases the experiments are conducted using the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL).

1.5 Origin of Chapters

Except for Chapter 2, the chapters in this thesis appeared before as refereed publications in international journals, and proceedings of conferences and workshops. Apart from the introduction (substantially extended) and Chapter 6 (major revision) only minor changes were applied before inclusion in this thesis. The origin of this thesis's chapters is as follows:

Chapter 1	Graaf, Bas. Model-driven evolution of software architectures. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 357–360. IEEE Computer Society, 2007

Chapter 3	Graaf, Bas, Marco Lormans, and Hans Toetenel. Embedded software engineering: The state of the practice. *IEEE Software*, 20(6):pages 61–69, 2003; and Graaf, Bas, Marco Lormans, and Hans Toetenel. Software technologies for embedded systems: An industry inventory. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES 2002)*, volume 2559 of *Lecture Notes in Computer Science*, pages 453–465. Springer-Verlag, 2002
Chapter 4	Graaf, Bas, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture – industrial experience report. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 354–363. IEEE Computer Society, 2005
Chapter 5	Van Dijk, Hylke W., Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, volume 3047 of *Lecture Notes on Computer Science*, pages 203–221. Springer-Verlag, 2005
Chapter 6	Graaf, Bas and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 115–126. IEEE Computer Society, 2007a
Chapter 7	Graaf, Bas, Sven Weber, and Arie van Deursen. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2007. Doi: 10.1016/j.jss.2007.06.007; and Graaf, Bas, Sven Weber, and Arie van Deursen. Migrating supervisory control architectures using model transformations. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 151–160. IEEE Computer Society, 2006
Chapter 8	Graaf, Bas and Arie van Deursen. Visualisation of domain-specific modelling languages using UML. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2007)*, pages 586–595. IEEE Computer Society, 2007c

Furthermore, our research has resulted in the following publications that are not directly included in this thesis:

- Graaf, Bas and Arie van Deursen. Using MDE for generic comparison of views. In *Proceedings of the 4th International Workshop on Model Design, Verification and Validation (MoDeVVa 2007)*, pages 57–66. INRIA, 2007b
- Spanjers, Hans, Maarten ter Huurne, Dan Bendas, Bas Graaf, Marco Lormans, and Rini van Solingen. Tool support for distributed software engineering. In *Proceedings of the 1st International Conference on Global Software Engineering (ICGSE 2006)*, pages 187–198. IEEE Computer Society, 2006
- Doyle, Duncan, Hans Geers, Bas Graaf, and Arie van Deursen. Migrating a domain-specific modeling language to MDA technology. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ateM 2006)*, number 1 / 2006 in Mainzer Informatik-Berichte, pages 47–54. Johannes Gutenberg-Universität Mainz, 2006
- Cornelissen, Bas, Bas Graaf, and Leon Moonen. Identification of variation points using dynamic analysis. In *Proceedings of the 1st International Workshop on Reengineering towards Product Lines (R2PL 2005)*, pages 9–13. 2005

Chapter 2

Background

In this chapter we elaborate on some of the concepts briefly introduced in Chapter 1. In particular, we discuss software evolution, software architecture, and model-driven engineering in the light of the research questions we posed previously.

2.1 Software Evolution

Engineering disciplines are typically based on universal, scientific laws and principles. Also in the discipline of software engineering a number of such laws have been discovered. Endres and Rombach [2003] give an overview. A few of the most widely acknowledged laws were defined by Lehman [1978] and are concerned with the change of software systems over time: software evolution. They are based on empirical observations. The first two of these, so-called, laws of software evolution dynamics are stated in Table 2.1 on the following page.

The graph in Figure 2.1 on the next page illustrates both laws by plotting a measure for size against a measure for complexity of embedded copier software developed by Océ. It shows a trend of increasing size and complexity for the subsequent (i.e., in time) revisions of the software.

Software evolves because of various reasons. The software systems (programs) referred to in the software evolution laws are, for instance, affected by changes in the reality reflected in their specification [Lehman, 1980]. Such changes are caused by changes in stakeholder objectives or to the environment. An example of the former are additional or modified stakeholder requirements. An example of the latter, in the case of embedded systems, are changes to hardware. As a response a software system requires *adaptive* maintenance. Obviously, the usefulness of a software system decreases if such maintenance tasks are not performed. Other types

Table 2.1: First two Laws of Software Evolution [Lehman, 1978]

I	*Law of Continuing Change* A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost-effective to replace the system with a recreated version.
II	*Law of Increasing Complexity* As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

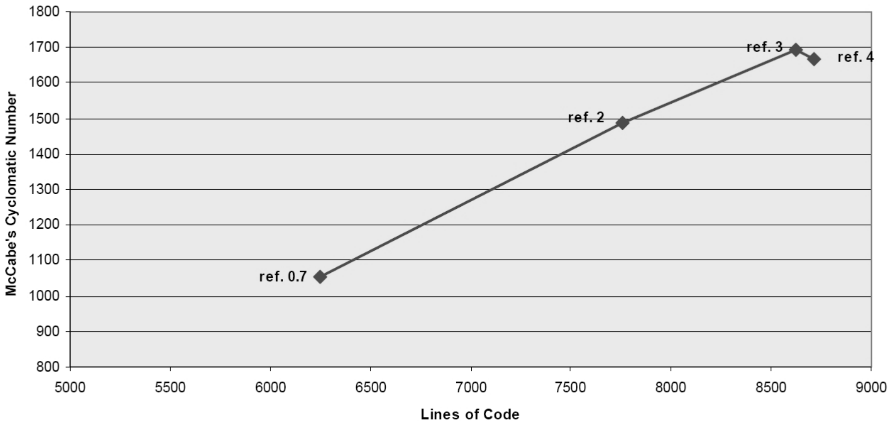


Figure 2.1: Complexity vs. size for subsequent revisions of copier software [Sonnenberg, 2005]

of maintenance tasks are *corrective* (removal of bugs) or *perfective* (optimising performance or maintainability) [Swanson, 1976; IEEE-1219, 1998] and also cause software to evolve. The first law states that such changes continue until it becomes more cost-effective to replace a system. However, development of replacement systems typically will not start from scratch, and significant parts of the already existing software will be reused. As such, the software continues to evolve.

Implicitly, the first law is based on the assumption that change becomes more expensive over the lifetime of a software system by stating that at some point replacement becomes more cost-effective than making changes. This is made explicit in the second law. It describes the unfortunate consequence of continual change: software systems become progressively more complex over time. In this law, complexity does not refer to computational complexity, but to the effort required to understand the inner workings of a software system. For a large part this effort depends on the structure of the software [Lehman, 1978], that is, its components and their relations. As change requires understanding, a consequence of increased complexity is that it makes a software system more difficult to change.

There are various explanations of this law of increasing complexity. Although, in theory, it might be possible to make changes to software systems without deteriorating its structure, practice is different. In industrial software projects, the users and customers of a software system are mainly concerned with its operation (e.g., performance, functionality), and not with its structure. This makes it difficult for the development organisation to justify longer lead-time of change requests because of structural preservation and recovery. Moreover, the effects of such efforts are not measurable immediately after changes are made, but are only long-term; they decrease the effort required for subsequent changes [Lehman, 1978]. Eventually, as this process of increasing complexity continues, it becomes infeasible to make even small changes to the software. Then, a system needs to be replaced.

Van Deursen [2005] proposed two possible strategies to deal with these laws: 1) postpone the moment at which a system needs to be replaced as much as possible by applying techniques to manage its ever-increasing complexity; and 2) apply techniques to restore the original structure or impose a new structure on the software system.

Considering the research questions posed in Chapter 1, we investigate in this thesis how and by the use of which technologies these two strategies can be supported. To this end we consider how specifications of software structure can be evaluated and manipulated. Furthermore, to take into account the complexity of software systems we investigate to what extent this can be automated. As an example, techniques to determine the consistency of different development artefacts, discussed in Chapters 5 and 6, help to manage complexity; techniques to automate the migration of a software architecture, discussed in Chapter 7, help to impose a new structure.

We already stated that a software system's complexity is strongly related to its structure (see Lehman's second law, Table 2.1 on page 18). The subfield of software engineering that studies software structure is called software architecture. The idea behind software architecture is that complexity can be managed by applying separation of concerns and abstraction. This is discussed in Section 2.2.

Another way to manage complexity is the use of automation. Model-driven engineering (MDE) is an approach to software development based on automation (and abstraction). With MDE applications are generated automatically by means of model transformations that transform abstract software models into source code. MDE is discussed in Section 2.3.

In this thesis we employ MDE techniques to support the evolution of software architectures. We conclude this chapter in Section 2.4 by explaining that due to a conceptual overlap MDE techniques are particularly suited for this purpose.

2.2 Architecture-Driven Software Development

2.2.1 Software Architecture

The development of a software system involves a large number of design decisions that eventually lead to an executable specification of its behaviour, typically in the form of source code. For a long time, it has been realised (e.g., by Dijkstra [1968], Parnas [1972] and Brooks, Jr [1975]) that, next to behaviour, it pays off to be also concerned with a software system's structure and organisation for reasons of dependability, understandability, and maintainability. Therefore, for large systems, these design decisions not only consider the *behaviour*, but also the *structures* of the software system. The key principles on which the design of software architectures is based are separation of concerns [Dijkstra, 1974] and abstraction.

Because of the complexity of software systems, multiple levels of abstraction are necessary to ensure designs remain comprehensible. This gives rise to several types of design. Usually, at least two levels of design are distinguished. Detailed design involves the decisions related to, for instance, data structures and algorithms. At a higher level of abstraction, design is called *software architecture* design [Garlan and Shaw, 1993], which is one of the key topics of this thesis.

It is difficult to capture the notion of software architecture in a single definition. As an example, the Software Engineering Institute [2006] collected many definitions. Perry and Wolf [1992] provide a model of software architecture consisting of elements, form, and rationale. The model distinguishes between three types of (design) elements: processing, data, and connecting elements. Form includes the relationships among the elements of an architecture. Rationale provides the motivation for the decisions that yield a particular set of elements and form. The three aspects of this model for software architecture can be found in various definitions for software architecture used by later research (and practice).

Garlan and Shaw [1993] enumerate a set of issues software architectures are concerned with that includes gross organisation, global control structure, communication protocols, and assignment of functionality to design elements.

A more recent definition of software architecture can be found in IEEE-1471 [2000]:

The fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

This definition not only includes components (elements) and their relations, but also principles, referring to, for instance, the use of a particular architectural style (see Section 2.2.3) or the use of particular conventions during design and maintenance of a software system.

An alternative definition that is frequently used is given by Bass et al. [2003]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

This definition acknowledges the now common understanding that there is no such thing as *the* structure of a software system and that different types of structures can be used to describe the architecture of a single system.

Kruchten [1998] states that software architecture encompasses a set of significant *decisions* regarding system organisation, selection of elements, their composition, and selection of an architectural style to guide these decisions. In this definition, architecture is thus considered as a set of decisions, a perspective further explored by Jansen [2005].

In summary, software architecture can be understood in at least two different ways: 1) as a set of (architectural) design decisions, or 2) as the structure that is the result of those decisions. In this thesis we opt for the latter, since we only consider software structures as prescribed by an architecture specification or as implemented in source code.

Unfortunately, through the use of adjectives like “significant”, “fundamental”, “gross”, and “global” the definitions cited above do not completely clarify which parts of a design are architectural and which parts are not.

Moreover, if a software architecture is a set of *architectural* design decisions, how do we determine whether a design decision is architectural? Eden et al. [2006] clarify this by providing a criterion that can be applied to design statements. The mathematically defined locality criterion states that a design statement is local if the system to which it applies cannot be made to violate it by mere expansion. Architecture statements are defined to belong to the class of non-local statements. For instance, the layered architectural style of a software system can be violated by simply expanding one of the layers with a component that interacts with components in non-adjacent layers. Hence, decisions regarding style are architectural. Conversely, a design pattern cannot be violated by only expanding a system. Thus, decisions regarding design patterns are not architectural.

Despite the precise definition discussed above, architecture is a relative concept because of the multiple levels of abstraction at which software design can be considered. What is architectural depends on, amongst others, the level of abstraction that is considered [Monroe et al., 1997; Clements et al., 2002a]: what is considered detailed design from a more abstract level can be considered architectural from a less abstract level.

From our collaborations with industry it became clear that in practice, architecture is 'defined' differently. There, different sets of decisions are considered to be architectural, for instance, the earliest (in time) decisions, the decisions that are most difficult (expensive) to change later on, or simply the decisions taken by the software architect. Although these sets might be slightly different, in this thesis we will assume they coincide, taking the point of view (as stated above) that an architecture is the result of such decisions.

2.2.2 Software Architecture Usage

So, why is it important to consider software architecture as a separate type of design? Bass et al. [2003] mention a few reasons. First, it allows to set apart the global design decisions, that is, those that affect multiple components, and hence need to be communicated to all involved developers to ensure the conceptual integrity [Brooks, Jr, 1975] of the system under development.

Second, as design decisions affect software quality attributes, a software architecture allows for early quality assessment of (to be developed) software systems. In fact, the software architecture is the first design artefact created in a software project that allows for such assessments.

Finally, software architectures enable the reuse of software solutions. By documenting a software architecture the design decisions it captures can be transferred to other systems, for instance, when similar quality attribute requirements need to be fulfilled.

These motivations for considering software architecture design as a separate phase in the software development process, also illustrates the importance of software evolution on the architectural level. If software is continuously evolved on lower abstraction levels, phenomena such as architecture drift and erosion [Perry and Wolf, 1992] (see also Chapter 1), decrease the possibility of using the software architecture in the ways described. At that point, restoration of the intended architecture or migration to a new architecture, again, brings the benefits of conceptual integrity, early assessment of design decisions, and reuse.

With respect to the preceding discussion on software architecture this thesis positions software architecture in terms of structure and architectural elements. Moreover, considering the importance of software structure for software evolution (see Section 2.1), we investigate how to use and manipulate software architectures for the software evolution tasks we identified in Section 1.1.

2.2.3 Software Architecture Design

The goal of software architecture *design* is to define the constraints for subsequent design and implementation activities that result in the development of a system that fulfils its functional and other quality goals. As such, a software architecture is both permissive and restrictive with respect to the decisions taken in subsequent activities [Perry and Wolf, 1992].

Based on existing design and evaluation methods, Kazman et al. [2006] formulate three principles that are useful to understand how architectural constraints are defined: 1) an architecture should be defined in terms of elements that are coarse enough for human intellectual control and specific enough for meaningful reasoning, 2) business goals determine quality attribute requirements, and 3) quality attribute requirements guide the design and analysis of software architectures.

Similar to other engineering disciplines, the actual design of software largely remains a creative activity. Consequently, the success of software projects for a large part depends on the experience and skills of the software architects. Although software architecture evaluation methods can help architects to assess the quality offered by the architecture they defined, such methods can only be applied *after* it has been designed. We introduce these methods in Section 2.2.5.

To also provide guidance during the design process itself, the analysis and codification of experiences by categorising problem types and recording and generalising successful (by experience) solutions, is of great importance for software architecture practice. Software architectural *styles*, sometimes also referred to as architectural patterns, are such codifications.

An architectural style is a set of constraints that is imposed on the architecture of systems that are based on that style. As such, an architectural style defines a set of architectures. The constraints defined by a style not only limit the type of architectural elements and their possible connections, but also dictate how their semantics should be interpreted [Abowd et al., 1993]. Shaw and Garlan [1996] and Buschmann et al. [1996], amongst others, provide collections of such architectural styles. The definition of architectural styles provides a shared vocabulary for software architects. Furthermore, it encourages researchers to study the properties of particular styles in terms of quality attributes. This gives way to architecture design (and evaluation) approaches that are based on the selection of appropriate styles for the desired quality attributes of a system, such as described by Klein et al. [1999] and by Bosch [2000].

Each style optimises a distinct set of quality attributes. In practice this implies the application of multiple architectural styles for the development of a single software system. As a result, multiple representations of such an architecture are conceivable; each clarifying a specific style.

2.2.4 Software Architecture Description

To effectively use (e.g., for evaluation or maintenance) the decisions that comprise a software architecture in non-trivial projects, it is required that these decisions are documented in a useful way. For the description of software architectures we distinguish between approaches based on: 1) architecture description languages (ADLs) (see Medvidovic and Taylor [1997] for an overview), and 2) views [IEEE-1471, 2000].

A large number of ADLs have been developed (mainly by the research community). Typically, such ADLs offer a formal syntax and semantics for the description of software architectures in terms of runtime components (computational elements) and connectors (abstractions for component interaction) [Medvidovic and Taylor, 1997]. As such, they allow to create precise descriptions of one aspect of a software system (i.e., its runtime structure and behaviour). Because of the formality of such descriptions, they can be used to automate several software engineering tasks, such as code generation and verification.

The ‘views approach’ on the other hand allows for more broad descriptions of software architecture. As discussed before, design can be considered on different *levels* of abstraction. However, we can also consider design with different *types* of abstractions. As such, a particular view might only consider runtime, which is typically the case with ADLs, or only design time aspects of a software system. The types of abstraction actually used can vary and are determined by what is important for a particular software project.

Views are based on the idea that a software architecture is too complex to be described in a single stroke, or by one type of abstraction (that is why we talked about structure*s* (plural) before). Multiple views are required to completely describe and document a software architecture. Each of those views addresses a specific set of concerns [IEEE-1471, 2000]. The guidelines for creating views are defined in so-called viewpoints, one for every type of view. Several sets of those viewpoints have been defined [Kruchten, 1995; Hofmeister et al., 1999; Clements et al., 2002a].

Compared to the 'ADL approach' this 'views approach' is more adopted by industry [Kruchten et al., 2006], where a view typically is a document that consists of some models or diagrams and explaining text, and is less formal than ADL-type descriptions of software architecture. When comparing the two approaches, it can be concluded that the ADL-approach as investigated by the research community focuses on in-depth description of software architectures, while the views-approach as used by industry focuses on broad description of software architectures [Medvidovic et al., 2002].

Finally, we specifically mention the Unified Modeling Language[1] (UML). Although UML was originally intended as a language for object-oriented modelling of systems, it has been used for architecture development as well (see, for instance, Chapter 3 and Lange et al. [2006]). To some extent it can be used as an ADL [Medvidovic et al., 2002]. Furthermore, UML diagrams are often used in architecture views.

In this thesis, we manipulate different types of architectural views to support software evolution. In some cases, we also partly demonstrate the definition of new ADLs. As an example, in Chapter 5 we transform one type of view into another to check the consistency of behavioural specifications of software embedded in the copiers developed by Océ.

2.2.5 Software Architecture Evaluation

An important reason for explicitly considering software architecture as a separate type of design activity or document, is that it constitutes the first opportunity for the prediction of properties of the system under development. We distinguish between two types of properties or qualities of a system: operational properties and development (i.e., non-operational) properties [Bosch, 2000]. The first type includes those properties that can be measured by observing the system in operation, such as functionality, performance, reliability. Non-operational properties, or development properties involve the development of the system, such as, maintainability, modifiability, portability, development cost and effort.

[1]http://www.uml.org (June 2007)

For the prediction of these properties different types of approaches have been developed [Bosch, 2000]. Approaches based on mathematical models, such as rate-monotonic analysis [Liu and Layland, 1973] and model checking [Clarke, Jr. et al., 1999], are best used for analysis of operational properties of the system. ADLs are also based on such models. On the other hand, scenario-based approaches such as, the Software Architecture Analysis Method (SAAM) [Kazman et al., 1996], the Architecture Tradeoff Analysis Method [Clements et al., 2002b] (ATAM), and architecture-level modifiability analysis [Bengtsson et al., 2004] (ALMA) (see also Dobrica and Niemelä [2002] for an overview of scenario-based software architecture analysis methods), are better suited for the analysis of development properties. These approaches use scenarios to make quality attribute requirements concrete after which the architecture is evaluated for its support for the identified scenarios. Such approaches can be used, for instance, to assess the maintainability of a software system.

In Chapter 4 of this thesis we will experiment with such an approach for the scenario-based evaluation of the maintainability of software embedded in the copiers developed by Océ.

2.2.6 Software Product Lines

One of the most important promises of the use of architectural principles for the development of software systems is the potential increase of reuse. This not only includes reuse of design decisions by capturing best practices in architectural styles, but also of architectural building blocks by explicitly defining what such components have to offer and what they rely on (i.e., their external visible properties).

The latter use, however, turned out to be problematic in practice because some degree of variation is typically required [Garlan et al., 1995]. To also account for variation and not only for commonalities, sets of similar software products can be viewed as software product families or product lines.

A product line encompasses a whole range of products that have much in common. By developing such products as a software product line [Clements and Northrop, 2002] their commonalities and variabilities are made explicit in a product-line architecture. The development of individual products is reduced to binding the variation points defined in the product-line architecture to specific instances, that is, if *all* variability is made explicit in the product-line architecture.

A software product line involves the development of product-line assets, such as a product-line architecture, reusable components, and product-line members. The assets that apply to the product line as a whole are developed in a process referred to as domain engineering. Product-line members are developed in a process called application engineering.

A product line can be ordered along a maturity scale by considering its (domain) scope, the extent that commonalities and variability are made explicit, and binding time of its variation points [Bosch, 2002]. A first step on this scale is the definition of all commonalities and their implementation as a (domain-specific) software platform. Product-line members are then built on top of that platform.

Both the systems that are the subject of our industrial case studies (the ASML wafer scanners and Océ copiers) are developed using product-line principles.

2.3 Model-Driven Engineering

To hide the structural and behavioural complexity of software platforms, approaches to software development have been introduced that are referred to as model-driven engineering (MDE) [Schmidt, 2006]. With MDE models are central instead of code. The idea is to develop software by transforming abstract models into more concrete models and eventually into code that typically runs on top of a software platform. Such transformations are referred to as model transformations. Because these transformations are automated, we are particularly interested in MDE technologies for the support of the software evolution tasks we defined. Furthermore, both MDE and software architectures are based on abstractions.

Some of the basic ideas behind MDE, that is, development of software by a series of model transformations and separation of functional specifications from the technical details of a specific platform, are very similar to that of stepwise refinement proposed by Wirth [1971]. Many topics related to MDE have been extensively studied by the research community: software reuse [Krueger, 1992], generative programming [Horowitz et al., 1985; Cleaveland, 1988; Czarnecki and Eisenecker, 2000], transformational programming [Partsch and Steinbrüggen, 1983], domain-specific languages [Van Deursen et al., 2000; Mernik et al., 2005], and environments and approaches for development of such languages [Klint, 1993; Van Deursen et al., 1996].

For a large part the development of current MDE approaches, however, has been driven by industry. Several implementations of the MDE concept are in use today. Often these are based on proprietary infrastructure that includes domain-specific (modelling) languages, application frameworks, code generators, and model repositories (see, e.g., Doyle et al. [2006]).

Additionally, a set of industry standards for MDE has been defined

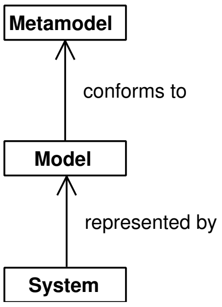


Figure 2.2: Fundamental relations between system, model, and metamodel [Bézivin, 2005]

by the Object Management Group[1] (OMG) under the name Model Driven Architecture[2] (MDA) and numerous tools now support part of these standards. Other, non-MDA tools are available that also support MDE, such as Microsoft's Domain-Specific Language Tools based on the approach by Greenfield et al. [2004]. It is the availability of these standards, supporting tools, and reusable software platforms that makes that current MDE approaches generate much more industrial momentum than earlier, related efforts [Schmidt, 2006].

Only recently, the research community has started investigating the fundamental principles behind MDE [Bézivin et al., 2007]. The foundations for MDE are abstraction (modelling) and automation (model transformations) [Sendall, 2003; Schmidt, 2006]. Modelling and the definition of the required modelling languages using, so-called, metamodels are based on the concepts and relations depicted in Figure 2.2. They are fundamental for MDE and are therefore introduced briefly in the following sections.

2.3.1 Modelling

Seidewitz [2003] defines a model as a set of statements about a system under study. Others have proposed similar definitions that add that a model has a specific purpose [Bézivin and Gerbé, 2001; OMG, 2007a]. A model can be used either descriptively to determine properties of a system, or prescriptively as a specification of a system to be built [Seidewitz, 2003]. The relation between a model and the system under study is referred to as represented by and was depicted in Figure 2.2.

For MDE approaches to be beneficial the involved models should be easier to create and understand than the systems they represent, and at the

[1]http://www.omg.org (June 2007)
[2]http://www.omg.org/mda (June 2007)

same time powerful enough to, for example, generate source code and perform assessments [Bézivin, 2005]. This requires that models leave out details that are irrelevant for their purpose. This simplification (or abstraction) is the essence of modelling [Bézivin and Gerbé, 2001].

Strictly speaking, source code is also a model [Mens and Van Gorp, 2006] that, with MDE, is the target of a model transformation. In practice, however, code and models are often considered to be different types of artefacts. Typically, in software engineering practice and in particular in the context of MDE something is considered to be a model if it has a graphical representation instead of only a textual one as in the case for source code [Mellor et al., 2003; Bézivin, 2006].

Kleppe et al. [2003] add another element to the definition of a model by stating that a model is written in a well-defined language. Thus, for the creation of models modelling languages are required. Basically two types of such languages exist: general-purpose languages (GPLs) and domain-specific languages (DSLs). UML is an example of a language in the MDA framework of the former type. It is applied in many different domains and for many different purposes. The latter type of languages is often specifically defined.

In this thesis the software evolution tasks we defined are applied to models as in the context of MDE. To this end, we not only use UML, but also (domain-specific) languages we defined ourselves using the MetaObject Facility[1] (MOF), the metamodelling language defined by the OMG.

2.3.2 Metamodelling

With MDE, the modelling languages used to create models are defined by metamodels. A metamodel is a graph composed of concepts and their relationships. From a usage perspective a metamodel determines which aspects of a system will be modelled in a corresponding model [Bézivin, 2006].

A metamodel is a model itself, that is, a model of a language. As such, a metamodel is, in turn, created using a modelling language. The metamodel used to define this metamodelling language is referred to as the metametamodel. This metametamodel is defined reflectively by using the language it defines itself. This yields a layered structure (architecture) of models as depicted in Figure 2.3 on the next page that is typical for MDE approaches. It is based on the two fundamental relations (represented by and conforms to) and concepts (system and model) of Figure 2.2 . For structures as in Figure 2.3 on the next page that model MDE concepts, such as system, model, and metamodel, the term megamodel was introduced [Bézivin et al., 2005; Favre, 2005b].

[1]http://www.omg.org/mof (June 2007)

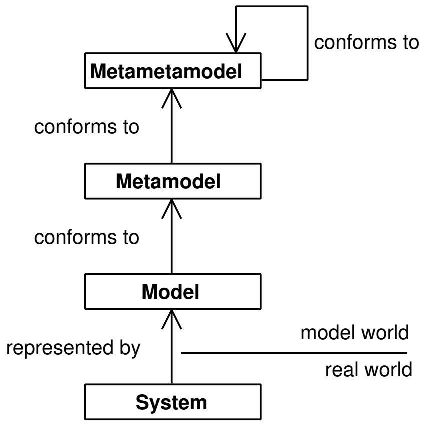


Figure 2.3: Layered MDE modelling stack

The relation between a model element and a corresponding metamodel element (i.e., on a higher model level) has to be distinguished from the instance-of relation that exists between a type and an instance of that type within the same modelling level. In practice these are often confused. As an example, consider Figure 2.4 . The metamodel is a simplified fragment of the UML metamodel. It defines a modelling language that allows to create models consisting of objects and classes that can be related to each other by the instance-of relation. These metamodel elements can be used to create a model of some application, for instance, to define a Class Person and an instance of that Class: Object joe. Both the relations between Class and Person and Person and joe are instance-of relations. However, they are of a different nature. Therefore, Atkinson and Küne [2003] distinguish between a linguistic (between Class and Person) and an ontological (between Person and joe) instance-of relation. Similarly, Bézivin and Gerbé [2001] uses a different term for the linguistic instance-of relation by referring to it as *meta*. Note that the conforms to relation depicted in Figure 2.3 effectively summarises *meta* relations between individual model and metamodel elements: a model conforms to a metamodel if and only if all its model elements have a *meta* relation with an element defined in that metamodel [Bézivin, 2006].

Favre [2005a] uses set theory to explain that the conforms to relation between a model and its metamodel is actually a derived relation as illustrated by Figure 2.5 . If we consider a modelling language as the set of models expressed in that language, the relation between a model and the modelling language is the element of relation from set theory. When we also consider the modelling language as the system under study, the modelling language is, in turn, related to the metamodel by the represented by relation.

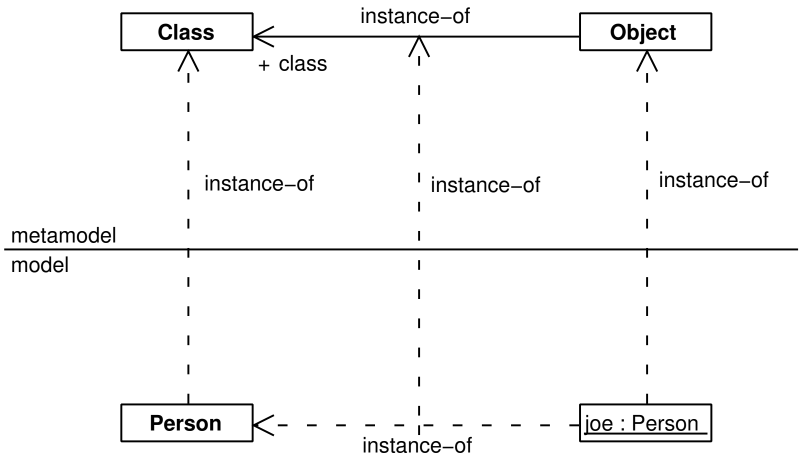


Figure 2.4: Metamodel and conforming model

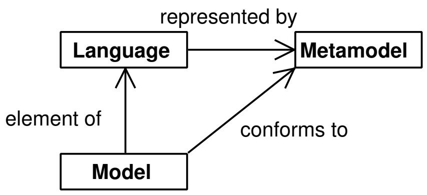


Figure 2.5: Conforms to relation in MDE

In the case of MDA, MOF is the metametamodel. MOF defines the (meta)model elements necessary to define modelling languages, such as, class, association, and constraint. Metamodelling is similar to data modelling and object modelling. As such, MOF models are similar to entity-relationship models [Chen, 1976] and, in particular, to UML class models.

Similarly to grammars, MOF is used to define the abstract syntax of modelling languages, that is, the structure of corresponding models. As an example, the UML metamodel is now also defined using MOF. Where, with (context-free) grammars, the abstract syntax defines a set of abstract syntax trees, a MOF metamodel defines a set of abstract syntax graphs. In contrast with grammars, MOF cannot be used to define the concrete syntax, that is, the notation, of a modelling language. In the case of UML, for instance, the notations used in the different diagrams are defined separately. In Chapter 8, we propose a light-weight solution to this problem.

Several implementations of MOF exist. These allow the definition of metamodels and the creation of conforming models in the Extensible Markup Language[1] (XML) or as objects in memory. Such implementations are used, for instance, for the development of model transformation tools. As an example, the Eclipse Modeling Framework[2] (EMF) is a plug-in for Eclipse[3] that is based on MOF. Given a metamodel EMF generates an implementation of that metamodel (in Java) that can be extended to develop tools based on that metamodel, such as a model editor.

For a particular MDE approach the metamodelling language can be used to define a whole class of modelling languages. Having a single metamodelling language also enables the development of model transformation languages and supporting tools.

In several of this thesis' chapters we created metamodels. In Chapter 6, for instance, we defined a simple modelling language to represent modularisation constructs, such as class and module; in Chapter 7, we defined a metamodel for the specification of task-resource models for control components in manufacturing machines.

2.3.3 Model Transformations

Model transformations are essential to MDE [Gerber et al., 2002; Sendall, 2003]. A transformation definition describes mappings or transformation rules to transform elements of a source model into elements of a target model [Kleppe et al., 2003]. Often model transformation languages use ex-

[1]http://www.w3.org/XML (June 2007)
[2]http://www.eclipse.org/emf (June 2007)
[3]Eclipse is a widely-used, freely-available, open-source integrated development environment, see http://www.eclipse.org (June 2007)

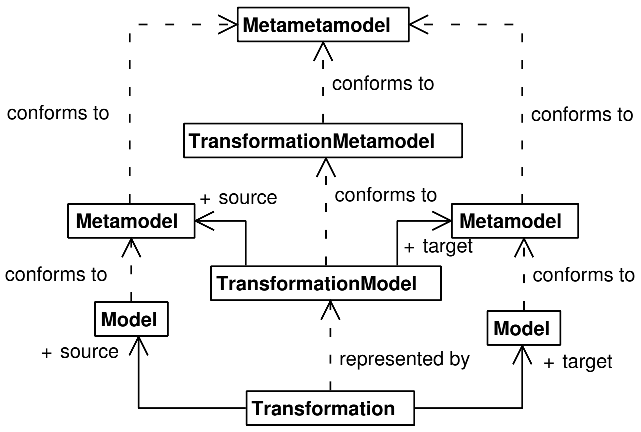


Figure 2.6: Model transformation megamodel

pressions in the Object Constraint Language[1] (OCL) to select the elements in the source model to transform. OCL is a declarative language, originally developed to specify constraints over UML models.

The MDE pattern or megamodel for model transformations is depicted in Figure 2.6 [Bézivin et al., 2005]. With MDE a Transformation between a source and target Model is defined by a transformation language. When this language is defined by a TransformationModelMetamodel, the transformation definition is in fact a model itself. This TransformationModel specifies transformations of source into target models in terms of the Metamodels they conform to. In correspondence with the metamodelling megamodel in Figure 2.3 on page 30, all involved metamodels conform to a single Metametamodel.

A transformation engine (automatically) transforms source models that conform to the source metamodel into target models that conform to the target metamodel as described in the transformation definition. As such, transformation engines require several inputs: source model, source metamodel, target metamodel, and transformation definition.

Many different types of model transformations and model transformation languages are conceivable. Sendall [2003]; Czarnecki and Helsen [2006]; and Mens and Van Gorp [2006] each give a number of properties of model transformation languages. These include the type and number of source and target models, horizontal vs. vertical transformations (with respect to abstraction level), type of notation (e.g., graphical vs. textual), source-target relationship (new vs. in-place), and many more.

[1]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)

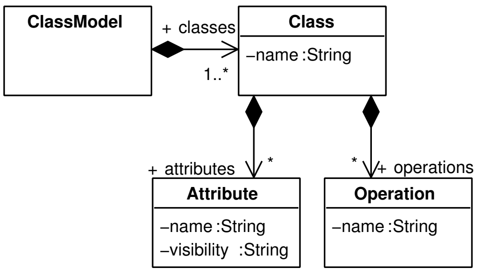


Figure 2.7: A metamodel for simple class models

ATL: a model transformation language In this thesis we mainly use the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL) to specify model transformations. Although, ATL is primarily a declarative model transformation language (based on OCL), it also has imperative features. An important reasons for using ATL is its implementation in a toolkit that includes an editor, debugger and transformation engine that is freely available. Furthermore, this toolkit integrates with EMF and other (UML) tooling. This allows the use of UML models as well as models based on custom (MOF-based) metamodels. Here, we briefly introduce ATL[1] as an example of a model transformation language.

As illustrative example, we discuss the transformation in Listing 2.1 that adds getters and setters to a simple class model. Figure 2.7 depicts a metamodel for simple class models. It contains a root element ClassModel that owns any number of classes. In turn, a Class, owns a number of attributes and operations. Class, Attribute, and Operation all have a name feature. Additionally, an Attribute also has a feature visibility.

Model transformations are specified in an ATL Module that is composed of a header, transformation rules and so-called helpers. As in Listing 2.1, the header specifies the names for the source models (`IN`), target models (`OUT`), and metamodels (`CLASSM`) that are used in the definition of transformation rules and helpers.

A declarative transformation rule is called a matched rule in ATL. It specifies a source and target pattern, indicated by the **`from`** and **`to`** blocks, respectively. The source pattern specifies the type of model elements that are matched by the rule. Optionally, the source pattern may specify a guard in the form of a Boolean OCL expression that further constraints the set of elements that are matched. For instance, in the `PrivateAttribute` rule (lines 31–40), the source pattern (`a_in`) specifies that the rule matches elements of type `Attribute` for which the `private` helper evaluates to true.

[1]Please, consult the ATL User Manual [ATLAS group] for more detailed information

```
module ADDGETSET;
create OUT : CLASSM from IN : CLASSM;

helper context CLASSM!Attribute def: isPrivate: Boolean =
 self.visibility = 'private';

rule ClassModel {
 from cm_in:CLASSM!ClassModel
 to
  cm_out:CLASSM!ClassModel (is
   classes <- cm_in.classes
 )
}
rule Class {
 from c_in:CLASSM!Class
 to
  c_out:CLASSM!Class (
   name <- c_in.name,
   attributes <- c_in.attributes,
   operations <- c_in.operations
    ->union(c_in.attributes->select(a|a.isPrivate)->collect(a|
     thisModule.resolveTemp(a,'get')))
    ->union(c_in.attributes->select(a|a.isPrivate)->collect(a|
     thisModule.resolveTemp(a,'set')))
 )
}
rule Attribute {
 from a_in:CLASSM!Attribute (not a_in.private)
 to
  a_out:CLASSM!Attribute (
   name <- a_in.name)
}
rule PrivateAttribute {
 from a_in:CLASSM!Attribute (a_in.private)
 to
  a_out:CLASSM!Attribute (
   name <- a_in.name),
  get:CLASSM!Operation (
   name <- 'get'+a_in.name),
  set:CLASSM!Operation (
   name <- 'set'+a_in.name)
}
rule Operation {
 from o_in:CLASSM!Operation
 to
  o_out:CLASSM!Operation (
   name <- o_in.name)
}
```

Listing 2.1: Adding getters and setters

The target pattern of a transformation rule may consist of multiple target pattern elements. Each element specifies the creation of a model element in the target model. A target pattern element specifies the type of the created element and a set of bindings that specifies how the features of the created model element will be initialised. The `PrivateAttribute` rule creates three elements in the target model by its target pattern elements: an Attribute (`a_out`) and two Operations (`get` and `set`). Their bindings refer to the matched source model element (`a_in`) to initialise the `name` feature of the created target model elements.

With ATL, the source model is read-only and the target model is write-only. To initialise the features of target model elements, the ATL transformation engine applies a specific value resolution algorithm. When the type of the expression of a binding is a primitive type (e.g., `c_in.name` in line 18) or when its value is another target pattern element of the same rule, it is simply assigned. When the value is a (set of) source model element(s) it is first resolved into a target model element (e.g., `c_in.attributes` in line 19). Effectively, the target model element created by the rule that matches the source model element specified in the binding's expression is assigned.

In the case that the matching rule specifies multiple target pattern elements the default (i.e., the first) target pattern element is used. When this is not desirable, ATL offers the `resolveTemp` operation that can be used to resolve source model elements using non-default target pattern elements. To this end, it takes the source model element to be resolved and the name of the target pattern element as parameters.

As an example, we explain how the operations feature of a target Class is initialised by the `Class` rule in lines 20–22. Using standard OCL operations the expression of the bindings specifies the union of the operations already present in the matching Class (`c_in.operations`) and the getters and setters generated for private Attributes. We specify the getters by first selecting from all attributes of the matching Class (`c_in.attributes`) those that are private using one of OCL's iterators (`->select(a|a.isPrivate )`). Subsequently, for the resulting Attributes we collect the created target elements using the `'get'` target pattern element of the matching rule (`->collect(a|thisModule.resolveTemp(a,'get')`). We specify the created setters similarly.

A helper may be defined in the context of some source metamodel element. As such, it effectively adds a feature to such an element. For example, the `isPrivate` helper (lines 4–5) is defined in the context of Attributes. It evaluates to true for every Attribute for which its visibility feature is set to the string `'private'`. This helper is an example of a so-called attribute helper. Such helpers do not take parameters and are evaluated only once for every source model element. Operation helpers, on the other hand, may take parameters and have to be evaluated upon each call. Finally, by omitting the context from the definition of a helper it is defined in the context of the transformation module as a whole, which is represented by the built-in `thisModule` element.

In addition to the (declarative) features explained above, ATL offers a number of additional features that we only mention briefly. Next to the simple target pattern elements in Listing 2.1 that generate a single target model element, iterative target pattern elements that iterate over some collection can be used to create a whole set of target model elements for a single matching source model element. ATL also offers imperative features that make it possible to add imperative instructions to matched rules. Finally, apart from matched rules, it is possible to define (imperative) rules that are not matched by source model elements, but that are called explicitly. As such, these *called* rules do not have a source pattern.

2.3.4 MDE and Other Technological Spaces

As discussed above, abstraction and automation are made possible in MDE by the use of metamodels. However, other solutions exist as well. As an example, modelling languages can also be defined using grammars or XML schemas. The layered structure depicted in Figure 2.3 on page 30 can be recognised when using these alternatives as well, for instance, in the case of grammars, EBNF is on the metametamodel level, grammars are on the metamodel level, and programs on the model level.

Such different types of solutions each come with a whole context of concepts, a body of knowledge, required skills, and possibilities. Kurtev et al. [2002] coined the term *technological space* for such a context. As such, the MDE space includes models, metamodels, model transformations, modelling tools and transformation languages. This space is also referred to as modelware. Similarly, the grammar space, or grammarware [Klint et al., 2005], includes programs, programming languages, grammars, parsers, and program transformation systems (e.g., Van den Brand et al. [2001]; Visser [2004]). Other recognised technological spaces are based on XML or ontologies, for instance.

Specific operations on a particular type of models might be more convenient in one technological space than in the other. For this reason it can be necessary to create a bridge between different technological spaces. In Chapter 6, for instance, we used a bridge between the modelware and grammarware technological space to visualise MDA models using a graph visualisation tool that has a grammar-based input language.

Another bridge that is very important to MDA is XML Metadata Interchange[1] (XMI). This bridge makes it possible to serialise models based on

[1]http://www.omg.org/mda/specs.htm#XMI (June 2007)

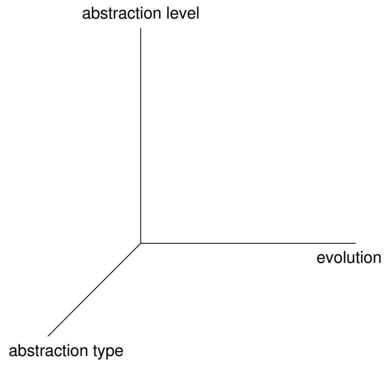


Figure 2.8: Three-dimensional evolution framework

MOF metamodels as XML documents. As such, XMI is used to exchange models between tools, for instance, between modelling and transformation tools.

In this thesis we mainly use technologies related to MDA, OMG's solution to MDE. It is a set of standards that includes capabilities for modelling (e.g., UML), metamodelling (MOF), and model transformations (Query/View/Transformation [OMG, 2005] (QVT)). The advantage of such standards is that they make it worthwhile for tool vendors to develop tools that support MDA technology [Booch et al., 2004], such as ArcStyler[1] (Interactive Objects) and Borland Together[2] Only when supporting tools are available an initiative as the MDA can become a success in practice. Tools that support UML modelling have been available for quite some time, but now also tools become available supporting metamodelling and model transformations[3].

2.4 Model-Driven Evolution of Software Architectures

In this thesis we investigate the model-driven evolution of software architectures. We conclude this chapter by explaining in this section why we think this makes sense.

Firstly, as explained in Section 2.2.4, architectures are often described using multiple views each addressing a specific set of concerns. For architectural views abstraction plays two roles: the level of abstraction, and the type of abstraction, which refers to the type of the view (i.e., the view-

[1]http://www.interactive-objects.com/products/arcstyler (September 2007)
[2]http://www.borland.com/us/products/together/ (September 2007)
[3]See http://planetmde.org/tools (June 2007) for an overview of MDA and MDE tools.

point [IEEE-1471, 2000]). Such a view is centred around a model. Often this is a model in a general sense, that is, it is a simplified representation of the system from a specific perspective. In practice such a model can be a drawing or sketch that is not based on a defined modelling language. In this thesis we attempt to consider those models in a more specific MDE sense, that is, models conforming to a metamodel. Using this perspective it becomes possible to support our software evolution tasks by model transformations.

By considering software evolution as driven by or the result of model transformations, we basically add a dimension along which models can be transformed to the two dimensions identified in Section 2.2.4 (type and level of abstraction). This results in a three-dimensional framework with two abstraction axes (one for type and one for level of abstraction) and one evolution axis. Models are transformed in a development (abstraction level) direction as well as in an (orthogonal) evolution direction. A third axis indicates the different types of abstractions (views) used (see Figure 2.8).

Secondly, the use of product-line principles can benefit especially from MDE approaches [Schmidt, 2006]. Software product lines are typically (at least) based on a platform [Bosch, 2002], a set of software components common to all product-line members. MDE approaches are particularly suited to be applied to generate code for such platforms by application of model transformations. We also apply these model transformations to support our evolution tasks in the case of an evolving product line architecture, or platform.

For these reasons this thesis explores the evolution (discussed in Section 2.1) of software architectures (Section 2.2) using model-driven techniques (Section 2.3).

Chapter 3

Embedded-Software Engineering: The State of the Practice[1]

The embedded-software market has grown very fast the last decade and will continue to do so in the coming years. The specific properties of embedded software, such as hardware dependencies, make its development different from non-embedded software. Therefore we expected very specific software development technologies to be used in this domain. The inventory we conducted at several embedded-software-development companies in Europe remarkably shows that this is not true. However the inventory results concerning requirements engineering and architecture design at these companies do suggest that there is a need for more specifically aimed development technologies. This chapter presents the inventory results and identifies possibilities for future research to customise existing and develop new software development technologies for the embedded-software domain.

3.1 Introduction

Many products today contain software (e.g., mobile telephones, DVD players, cars, aeroplanes, and medical systems). Because of advancements in information and communication technology, in the future even more products will likely contain software. The market for these 'enhanced' products is forecasted to grow exponentially in the next 10 years [Progress, 2002]. Moreover, these embedded-systems' complexity is increasing, and the amount and variety of software in these products are growing. This creates a big challenge for embedded-software development. In the years

[1]This chapter was published earlier as: Graaf, Bas, Marco Lormans, and Hans Toetenel. Embedded software engineering: The state of the practice. *IEEE Software*, 20(6):pages 61–69, 2003

to come, the key to success will be the ability to successfully develop high-quality embedded systems and software on time. As the complexity, number, and diversity of applications increase, more and more companies are having trouble achieving sufficient product quality and timely delivery. To optimise the timeliness, productivity, and quality of embedded-software development, companies must apply software engineering technologies that are appropriate for specific situations.

Unfortunately, the many available software development technologies don't take into account the specific needs of embedded-systems development. This development is fundamentally different from that of non-embedded systems. Technologies for the development of embedded systems should address specific constraints such as hard timing constraints, limited memory and power use, predefined hardware platform technology, and hardware costs. Existing development technologies don't address their specific impact on, or necessary customisation for, the embedded domain. Nor do these technologies give developers any indication of how to apply them to specific areas in this domain – for example, automotive systems, telecommunications, or consumer electronics. Consequently, tailoring a technology for a specific use is difficult. Furthermore, the embedded domain is driven by reliability factors, cost factors, and time to market. So, this embedded domain needs specifically targeted development technologies.

In industry, the general feeling is that the current practice of embedded-software development is unsatisfactory. However, changes to the development process must be gradual; a direction must be supplied. To achieve this, we need more insight into the currently available and currently used methods, tools, and techniques in industry.

To gain such insight, we performed an industrial inventory as part of the Software Engineering *M*eth*O*d*O*logie*S* for *E*mbedded Systems[1] (MOOSE) project. MOOSE is part of the Information Technology for European Advancement[2] (ITEA) programme and is aimed at improving software quality and development productivity for embedded systems. Not only did we gain an overview of which technologies the MOOSE consortium's industrial partners use, we also learnt why they use or don't use certain technologies. In addition, we gained insight into what currently unavailable technologies might be helpful in the future.

3.2 Methods and Scope

The inventory involved seven industrial companies and one research institute in three European countries (see Table 3.1). These companies build a

[1]http://www.mooseproject.org (June 2007)
[2]http://www.itea-office.org (June 2007)

Table 3.1: Inventoried companies

Company	**Products**
TeamArteche (Spain)	Measurement, control, and protection systems for electrical substations
Nokia (Finland)	Mobile networks and mobile phones
Solid (Finland)	Distributed-data-management solutions
VTT Electronics (Finland)	Technology services for businesses
Philips PDSL (Netherlands)	Consumer electronics
ASML (Netherlands)	Lithography systems for the semiconductor industry
Océ (Netherlands)	Document-processing systems
LogicaCMG (Netherlands)	Global IT solutions and services

variety of embedded-software products, ranging from consumer electronics to highly specialised industrial machines. We performed 36 one-hour interviews with software practitioners. The respondents were engineers, researchers, software or system architects, and managers, with varying backgrounds. To get a fair overview of the companies involved (most of which are very large), we interviewed at least three respondents at the smaller companies and five or six at the larger companies. These interviews were conducted in the period April – October 2002.

We based the interviews on an outline specifying the discussion topics (see Table 3.2 on the following page). To be as complete as possible, it is based on a reference process model. Because software process improvement methods have such a (ideal) process model as their core, we used one of them. We chose the process model of the BOOTSTRAP method [Kuvaja et al., 1994] because of its relative emphasis on engineering processes compared to other process models [Wang et al., 1999], such as those of the Capability Maturity Model [Humphrey, 1989] (CMM) and Software Process Improvement and Capability dEtermination [Emam et al., 1997] (SPICE). BOOTSTRAP's other advantage for this inventory is that the BOOTSTRAP Institute developed it with the European software industry in mind.

For every interview we created a report; we consolidated the reports for a company into one report. We then analysed the company reports for trends and common practices. Finally, we wrote a comprehensive report that, for confidentiality reasons, is available only to MOOSE consortium members. That report forms the basis for this discussion.

Table 3.2: A sample of the interview outline

Here are some discussion topics and questions from the outline we used for the interviews.

Technology

What are the most important reasons for selecting development technologies?

- Impact of introducing new technologies (cost, time, and so on).
- Why not use modern/different technologies?

Software life cycle

Software requirements engineering
How are the requirements being gathered?

- What are the different activities?
- What documents are produced?
- What about tool support?

How are the requirements being specified?

- What specification language?
- What about tool support? (Consider cost, complexity, automation, training, acceptance)
- What notations/diagrams?
- What documents are produced?
- How are documents reviewed?
- What are advantages/disadvantages of followed approaches?

Software architecture design
How is the architecture specified?

- What architecture description language?
- What about tool support? (Consider cost, complexity, automation, training, acceptance)
- Are design patterns used?
- What notations/diagrams?
- What documents are produced?
- How are documents reviewed?
- What are advantages/disadvantages of followed approaches?

3.3 Embedded-Software Development Context

When considering the embedded-software-development process, you need to understand the context in which it is applied. After all, most companies that develop embedded software do not sell it. Although at the time of writing this is slowly changing in some industries (e.g., consumer electronics, see Van Genuchten [2007], they primarily sell mobile phones, CD players, lithography systems, and other products. The software in these products constitutes only one (important) part. Embedded-software engineering and other processes such as mechanical engineering and electrical engineering are in fact subprocesses of systems engineering. Coordinating these subprocesses to develop quality products is one of embedded-system development's most challenging aspects. The increasing complexity of systems makes it impossible to consider these disciplines in isolation.

For instance, when looking at communication between different development teams, we noticed that besides vertical communication links along the lines of the hierarchy of architectures, horizontal communication links existed. Vertical communication occurs between developers who are responsible for systems, subsystems, or components at different abstraction levels (e.g., a system architect communicating with a software architect). Horizontal communication occurs between developers who are responsible for these things at the same abstraction level (e.g., a programmer responsible for component A communicating with a programmer responsible for component B).

Still, we found that systems engineering was mostly hardware driven – that is, from a mechanical or an electronic viewpoint. In some companies, software architects weren't even involved in design decisions at the system level. Hardware development primarily dominated system development because of longer lead times and logistical dependencies on external suppliers. Consequently, software development started when hardware development was already at a stage where changes would be expensive. Hardware properties then narrowed the solution space for software development. This resulted in situations where software inappropriately fulfilled the role of integrator; that is, problems that should have been solved in the hardware domain were solved in the software domain. Embedded-software developers felt that this was becoming a serious problem. So, in many companies this was changing; software architects were becoming more involved on the system level.

Depending on the product's complexity, projects used system requirements to design a system architecture containing multidisciplinary or monodisciplinary subsystems. (A multidisciplinary subsystem will be implemented by different disciplines; a discipline refers to software, or mechanics, or electronics, or optics, and so on. A monodisciplinary subsys-

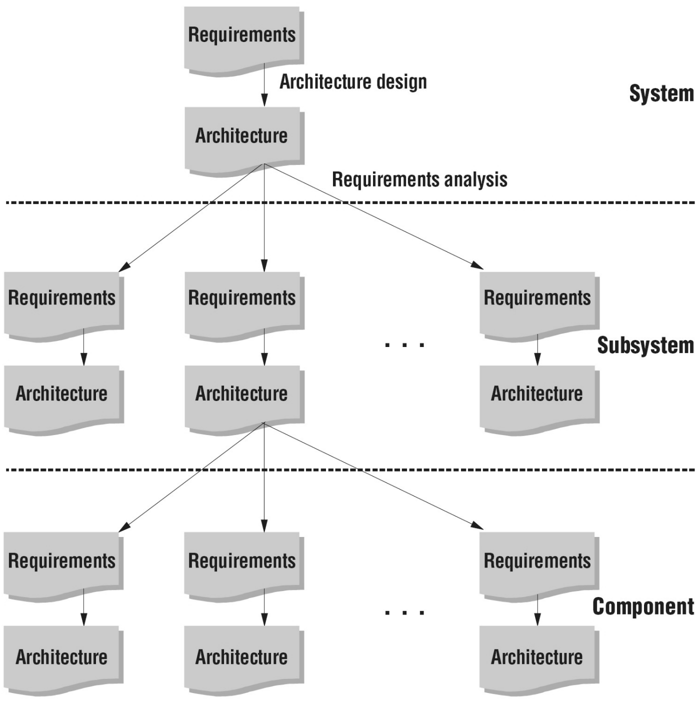


Figure 3.1: The decomposition of the embedded-systems-development process

tem will be implemented by one discipline.) Next, the projects allocated system requirements to the architecture's different elements and refined the requirements. This process is repeated for each subsystem. Finally, the projects decomposed the subsystems into monodisciplinary components that an individual developer or small groups of developers could develop. The level of detail at which decomposition resulted in monodisciplinary subsystems varied. In some cases, the first design or decomposition step immediately resulted in monodisciplinary subsystems and the corresponding requirements. In other cases, subsystems remained multidisciplinary for several design steps.

This generic embedded-systems-development process resulted in a tree of requirements and design documents (see Figure 3.1). Each level represented the system at a specific abstraction level. The more complex the system, the more evident this concept of abstraction levels was in the development process and its resulting artefacts (for example, requirements documentation).

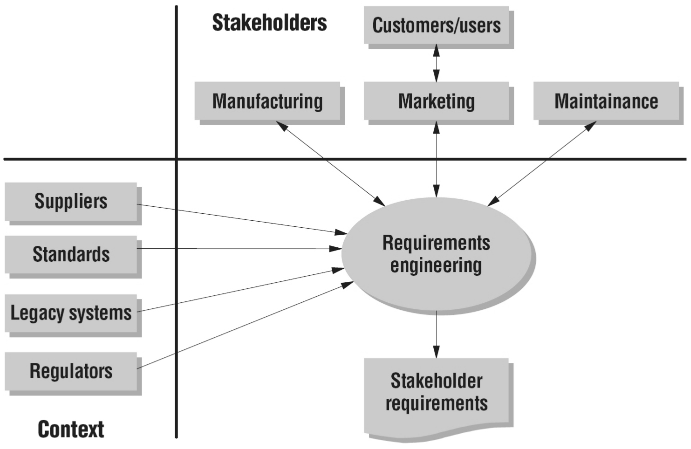


Figure 3.2: Embedded-systems-development stakeholders and other factors

In the process in Figure 3.1 , requirements on different abstraction levels are related to each other by design decisions, which were recorded in architecture and design documentation. At the system level, these decisions concerned partitioning of the functional and nonfunctional requirements over software and hardware components. The criteria used for such concurrent design (codesign) were mostly implicit and based on system architects' experience.

3.4 Requirements Engineering Results

Typically, embedded-systems development involved many stakeholders. This was most apparent during requirements engineering. Figure 3.2 depicts our view of the most common stakeholders and other factors.

In requirements engineering's first phase, the customer determines the functional and nonfunctional requirements. Depending on the product domain, the customer negotiates the requirements via the marketing and sales area or directly with the developers.

The first phase's output is the agreed requirements specification, which is a description of the system that all stakeholders can understand. This document serves as a contract between the stakeholders and developers. At this point, we noticed a clear difference between small and large projects. In small projects, the stakeholder requirements also served as developer requirements. In large projects, stakeholder requirements were translated into technically oriented developer requirements.

Requirements specify what a system does; a design describes how to realise a system. Software engineering textbooks strictly separate the requirements and the design phases of software development; in practice, this separation is less obvious. In fact, the small companies often put both the requirements and design into the system specification. These companies did not explicitly derive software requirements from the system requirements. The development processes in the larger companies did result in separate requirements and design documents on different abstraction levels. However, in many cases, these companies directly copied information from a design document into a requirements document for the next abstraction level instead of first performing additional requirements analysis. For instance, a software architecture specification (i.e., a design document) might list some characteristics of the components that comprise the architecture. On the next abstraction level a requirements document concerns only an individual component. Often simply the characteristics mentioned in the architecture specification are used as the requirements, instead of elaborating those requirements and adding more detailed requirements.

3.4.1 Requirements Specification

Requirements were usually specified in natural language and processed with an ordinary word processor. The companies normally used templates and guidelines to structure the documents. The templates prescribed what aspects had to be specified. However, not all projects at a company used these templates, so requirements specifications from different projects sometimes looked quite different.

Because embedded-software's nonfunctional properties are typically important, we expected these templates to reserve a section on nonfunctional requirements next to functional requirements. This wasn't always the case. For example, the requirements specification didn't always explicitly take into account real-time requirements. Sometimes a project expressed them in a separate section in the requirements documents, but often they were implicit. Requirements specification and design also usually didn't explicitly address other typical embedded-software requirements, such as those for power consumption and memory use.

Projects that employed diagrams to support requirements used mostly free-form and box-line diagrams in a style that resembles the Unified Modeling Language[1] (UML), data-flow diagrams, or other notations. Project members primarily used general-purpose drawing tools to draw the diagrams. Because of the lack of proper syntax and semantics, other project members often misinterpreted the diagrams. This was especially true for

[1]http://www.uml.org (June 2007)

project members working in other disciplines that employ a different type of notation.

UML was not common practice yet, but most companies were at least considering its possibilities for application in requirements engineering. Use cases were the most-used UML constructs in this phase. Some projects used sequence diagrams to realise use cases; others applied class diagrams for domain modelling. However, the interpretation of UML notations was not always agreed on during requirements engineering. It wasn't always clear, for instance, what objects and messages in UML diagrams denote when a sequence diagram specifies a use case realisation.

On the lowest levels, projects commonly used pre- and postconditions to specify software requirements. They specified interfaces as pre- and postconditions in natural language, C, or some interface definition language.

Projects rarely used formal specifications. One reason was that formal specifications were considered difficult to use in complex industrial environments and require specialised skills. When projects did use them, communication between project members was difficult because most members did not completely understand them. This problem worsened as projects and customers needed to communicate. In one case, however, a project whose highest priority was safety used the formal notation Z for specification.

3.4.2 Requirements Management

When looking at Figures 3.1 on page 46 and 3.2 on page 47, you can imagine that it's hard to manage the different requirements from all these different sources throughout development. This issue was important especially in large projects.

Another complicating factor was that most projects didn't start from scratch. In most cases, companies built a new project on previous projects. So, these new projects reused requirements specifications (even for developing a new product line). Consequently, keeping requirements documents consistent was difficult. To keep all development products and documents consistent, the projects had to analyse the new features' impact precisely. However, the projects frequently didn't explicitly document relations between requirements, so impact analysis was quite difficult. This traceability is an essential aspect of requirements management. Tracing requirements was difficult because the relations (e.g., between requirements and architectural components) were too complex to specify manually.

Available requirements management tools didn't seem to solve this problem, although tailored versions worked in some cases. A general shortcoming of these tools was that the relations between the requirements had no meaning. In particular, tool users could specify the relations but not the rationale behind the link.

When projects did document relations between requirements, they used separate spreadsheets. Some companies were using or experimenting with more advanced requirements management tools such as RequisitePro (Rational), RTM (Integrated Chipware), and DOORS (Telelogic). These experiments weren't always successful. In one case, the tool's users didn't have the right skills, and learning them took too long. Also, the tool handled only the more trivial relations between requirements, design, and test documents. So, developers couldn't rely on the tool completely, which is important when using a tool.

Requirements management also involves release management (managing features in releases), change management (backwards compatibility), and configuration management. Some requirements management tools supported these processes. However, because most companies already had other tools for this functionality, integration with those tools would have been preferable.

3.5 Software Architecture Results

Small projects didn't always consider the explicit development, specification, and analysis of the product architecture necessary. Also, owing to time-to-market pressure, the scheduled deadlines often obstructed the development of sound architectures. Architects often said they didn't have enough time to do things right.

The distinction between detailed design and architecture seemed somewhat arbitrary. During development, the projects interpreted architecture simply as high-level design. They didn't make the distinction between architectural and other types of design explicit, as, for example, Eden and Kazman [2003]. There, the locality criterion is introduced to distinguish architectural design from detailed design. A design statement is said to be local when it can't be violated by mere expansion. The application of a design pattern is an example of a local design statement. Architectural design is not local. For instance, an architectural style can be violated by simple expansion.

3.5.1 Software Architecture Design

Designing a product's or subsystem's architecture was foremost a creative activity that was difficult to divide into small, easy-to-take steps. Just as system requirements formed the basis for system architecture decisions, system architecture decisions constrained the software architecture.

In some cases, a different organisational unit had designed the system architecture. So, the architecture was more or less fixed – for instance, when the hardware architecture was designed first or was already known. This led to suboptimal (software) architectures. Because software was considered more flexible and has a shorter lead time, projects used it to fix hardware architecture flaws, as we mentioned before.

The design process didn't always explicitly take into account performance requirements. In most cases where performance was an issue, projects just designed the system to be as fast as possible. They didn't establish how fast until an implementation was available. Projects that took performance requirements into account during design did so mostly through budgeting. For example, they frequently divided a high-level real-time constraint among several lower-level components. This division, however, often was based on the developers' experience rather than well-funded calculations. Projects also used this technique for other non-functional requirements such as for power and memory use.

Projects sometimes considered commercial off-the-shelf (COTS) components as black boxes in a design, specifying only the external interfaces. This was similar to an approach that incorporated hardware drivers into an object-oriented (OO) design. However, developers of hardware drivers typically don't use OO techniques. By considering these drivers as black boxes and looking only at their interfaces, the designers could nevertheless include them in an OO design. For the COTS components, the black box approach wasn't always successful. In some cases, the projects also had to consider the components' bugs, so they couldn't treat the components as black boxes.

The software architecture often mirrored the hardware architecture, which made the impact of changes in hardware easier to determine. Most cases involving complex systems employed a layered architecture pattern. These layers made it easier to deal with embedded-systems' growing complexity.

3.5.2 Software Architecture Description

UML was the most commonly used notation for architectural modelling. On the higher abstraction levels, the specific meaning of UML notations in the architecture documentation should be clear to all stakeholders, which was not always the case. Some projects documented this in a reference architecture or architecture manual (we discuss these documents in more detail later).

IBM's Rational Rose Technical Developer[1] (formerly known as Rational Rose RealTime) lets developers create executable models and completely generate source code. A few projects tried this approach. One project completely generated reusable embedded-software components from Rational Rose RealTime models. However, most of these projects used these tools only experimentally.

For creating UML diagrams, respondents frequently mentioned only two tools: Microsoft Visio and Rational Rose. Projects used these tools mostly for drawing rather than modelling. This means, for instance, that models weren't always syntactically correct and consistent.

Other well-known notations that projects used for architectural modelling were data-flow diagrams, entity-relationship diagrams, flowcharts, and Hatley-Pirbhai diagrams [Hatley and Pirbhai, 1987] for the representation of control flow and state-based behaviour. Projects often used diagrams based on these notations to clarify textual architectural descriptions in architecture documents. Some projects used more free-form box-line drawings to document and communicate designs and architectures.

One project used the Koala component model [Van Ommering et al., 2000] to describe the software architecture. Compared to box-line drawings, the Koala component model's graphical notation has a more defined syntax. Koala provides interface and component definition languages based on C syntax. A Koala architecture diagram specifies the interfaces that a component provides and requires. This project used Microsoft Visio to draw the Koala diagrams.

Projects often used pseudocode and pre- and postconditions to specify interfaces. Although this technique is more structured than natural language, the resulting specifications were mostly incomplete, with many implicit assumptions. This not only sometimes led to misunderstandings but also hampered the use of other techniques such as formal verification.

Some projects referred to a reference architecture or an architecture user manual. These documents defined the specific notations in architectural documents and explained which architectural concepts to use and how to specify them.

3.5.3 Software Architecture Evaluation

Most projects did not explicitly address architecture verification during design; those that did primarily used qualitative techniques. Few projects used quantitative techniques such as Petri nets or rate monotonic scheduling analysis [Liu and Layland, 1973]. One reason is that quantitative-analysis tools need detailed information. In practice, projects often used an

[1]http://www-306.ibm.com/software/awdtools/developer/technical (June 2007)

architecture only as a vehicle for communication among stakeholders.

The most commonly employed qualitative techniques were reviews, meetings, and checklists. Another qualitative technique employed was scenario-based analysis. With this technique, a project can consider whether the proposed architecture supports different scenarios. By using different types of scenarios (e.g., use scenarios and change scenarios), a project not only can validate that the architecture supports a certain functionality but also can verify qualities such as changeability.

The respondents typically felt that formal verification techniques were inapplicable in an industrial setting. They considered these techniques to be useful only in limited application areas such as communication protocols or parts of security-critical systems. The few projects that used Rational Rose RealTime were able to use simulation to verify and validate architectures.

3.5.4 Reuse

Reuse is often considered one of the most important advantages of development using architectural principles. By defining clean, clear interfaces and adopting a component-based development style, projects should be able to assemble new applications from reusable components.

In general, reuse was rather ad hoc. Projects reused requirements, design documents, and code from earlier, similar projects by copying them. This was because most products were based on previous products.

For highly specialised products, respondents felt that using configurable components from a component repository was impossible. Another issue that sometimes prevented reuse was the difficulty of estimating both a reuse approach's benefits and the effort to introduce it.

In some cases a project or company explicitly organised reuse. One company did this in combination with the Koala component model. The company applied this model together with a proprietary method for developing product families.

Some companies had adopted a product-line approach to create a product line or family architecture. When adopting this approach, the companies often had to extract the product-line architecture from existing product architectures and implementations. This is called *reverse architecting*.

In most cases, hardware platforms served as the basis for defining product lines, but sometimes market segments determined product lines. When a company truly followed a product-line approach, architecture design took variability into account.

One company used a propriety software development method that enabled large-scale, multisite, and incremental software development. This method defined separate long-term architecture projects and subsystem projects. The company used the subsystems in short-term projects to instantiate products.

Another company had a special project that made reusable components for a certain subsystem of the product architecture. The company used Rational Rose RealTime to develop these components as executable models.

Some companies practised reuse by developing general platforms on top of which they developed different products. This strategy is closely related to product lines, which are often defined per platform.

3.6 Discussion

You might well ask, are these survey results representative of the whole embedded-software domain? By interviewing several respondents with different roles in each company, we tried to get a representative understanding of that company's embedded-software-development processes. The amount of new information gathered during successive interviews decreased. So, we concluded we did have a representative understanding for that company.

With respect to embedded-software development in general, we believe that the large number of respondents and the companies' diversity of size, products, and country of origin make this inventory's results representative, for Europe at least. However, whether we can extend these results to other areas (e.g., the United States) is questionable.

Another point for discussion is that the methods, tools, and techniques the companies used were rather general software engineering technologies. We expected that the companies would use more specialised tools in this domain. Memory, power, and real-time requirements were far less prominent during software development than we expected. That's because most general software engineering technologies didn't have special features for dealing with these requirements. Tailoring can be a solution to this problem, but it involves much effort, and the result is often too specific to apply to other processes. Making software development technologies more flexible can help make tailoring more attractive. So, flexible software development technologies are necessary. Here, with flexible we mean, for instance, requirements management tools that allow to modify the types and characteristics of the managed requirements, or model transformation tools that allow to transform models in arbitrary modelling languages, instead of being restricted to UML.

We noticed a relatively large gap between the inventory's results and the available software development technologies. Why isn't industry using many of these technologies? During the interviews, respondents mentioned several reasons. We look at three of them here.

The first reason is compliance with legacy. As we mentioned before, most projects didn't start from scratch. Developers always have to deal with this legacy, which means that the technology used in current projects should at least be compatible with the technology used in previous products. Also, companies can often use previous products' components in new products with few or no adaptations. This contradicts the top-down approach in Figure 3.1 on page 46. Unlike with that approach, components at a detailed level are available from the start, before the new product's architecture is even defined. This would suggest a bottom-up approach. However, because most available software development approaches are top-down, they don't address this issue.

Another reason is maturity. Most development methods are defined at a conceptual level; how to deploy and use them is unclear. When methods are past this conceptual stage and even have tool implementations, the tools' maturity can still prevent industry from using them. This was the case for some requirements management tools. Some respondents said that these tools weren't suited for managing the complex dependencies between requirements and other development artefacts, such as design and test documentation. Also, integrating these tools with existing solutions for other problems such as configuration management and change management was not straightforward.

The third reason is complexity. Complex development technologies require highly skilled software engineers to apply them. But the development process also involves stakeholders who aren't software practitioners. For instance, as we mentioned before, project team members might use architecture specifications to communicate with (external) stakeholders. These stakeholders often do not understand complex technology such as formal architecture description languages (ADLs). Still, formal specifications are sometimes necessary – for example, in safety-critical systems. To make such highly complex technologies more applicable in industry, these technologies should integrate with more accepted and easy-to-understand technologies. Such a strategy will hide complexity.

3.7 Outlook

In the remainder of this thesis we take into account the aforementioned reasons for industry's reluctance of adopting state-of-the-art software development technologies. This implies that we, were possible, make use of existing standards and technologies that already have been successfully applied in industrial practice.

Moreover, we have seen that software development in practice seldom starts from scratch. As such, technologies to support software maintenance deserve at least as much attention as those that support software development. Therefore, we focus on software evolution and how related software engineering tasks can be supported. To this end, we use and combine existing software engineering technologies as much as possible.

The trend we observed that software development is moving towards larger scale and more structured reuse by the use of software product-line approaches is a final important consideration in the remainder of this thesis.

Chapter 4

Evaluating an Embedded Software Reference Architecture – Industrial Experience Report –[1]

In this chapter, we discuss experiences gained during evaluation of the maintainability of a software reference architecture in use at Océ, one of the world's leading copier manufacturers. The evaluation is conducted using an approach based on the Software Architecture Analysis Method. The chapter proposes a variant of this method that helps to reduce the organisational impact of architecture evaluations. Second, we analyse the implications of evaluating reference architectures as opposed to single-product architectures. Furthermore, we share our experience of conducting the evaluation, draw lessons for practitioners, and propose new research topics.

4.1 Introduction

In industry new products are rarely developed from scratch. Most products are based on previous generations of similar products. Therefore, the capability of reusing large parts of earlier development efforts when developing new products can increase the development efficiency of companies tremendously [Jacobson et al., 1997]. However, currently many companies have no structured approach for reuse, as the inventory conducted among several companies developing embedded software confirmed (see Chapter 3).

[1]This chapter was published earlier as: Graaf, Bas, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture – industrial experience report. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 354–363. IEEE Computer Society, 2005

One strategy to arrive at structured reuse, is to adopt architectural concepts, including product-line approaches, during the software development process. Architecture-based development increases development efficiency and makes software systems more easy to maintain and evolve. It does so by increasing the conceptual integrity [Brooks, Jr, 1975] of software systems and by providing a common software infrastructure which makes it easier to understand systems and to integrate new components. A product-line architecture extends these ideas beyond single-product developments to a whole generation of products and thus enables the reuse of components in new product-line members.

At Océ, one of the world's leading copier manufacturers, every couple of years a new product generation is launched, comprising a family of similar products. To make development and maintenance of these generations more effective and efficient Océ decided to define a reference architecture for a part of the embedded software in its products. It establishes a common software infrastructure for different generations, thus facilitating reuse across generation boundaries.

Since this reference architecture will potentially impact all embedded software to be developed at Océ, the architecture team at Océ decided to conduct an evaluation of the quality of this reference architecture, using an approach based on the Software Architecture Analysis Method (SAAM) [Kazman et al., 1996; Clements et al., 2002b] which was developed at the Software Engineering Institute[1] (SEI). In this chapter we report on this evaluation.

The contributions of this chapter are threefold. First, we propose a variant of SAAM that reduces the organisational impact of architecture evaluations. Second, we analyse the implications of evaluating reference architectures as opposed to product architectures. Last but not least, we share our experience with conducting an evaluation of a real-life reference architecture that is actually used in industry. The lessons learnt are useful for practitioners, and lead to new research questions related to architecture evaluation.

In order to protect Océ's interests, we cannot discuss Océ-sensitive details of the reference architecture. Instead, we will discuss a modified version. We believe that the architectural issues and the evaluation method are not materially affected by these changes.

This chapter is organised as follows. In Section 4.2 we summarise the content and context of the embedded software reference architecture for copier engines (hereafter referred to as 'the reference architecture'). In Section 4.3, we describe why we selected SAAM to conduct the evaluation and why Océ's situation required some modifications to it. In Section 4.4

[1]http://www.sei.cmu.edu (June 2007)

we explain how the actual evaluation was carried out and how practical problems were solved. Then, in Section 4.5 we reflect on the evaluation and identify future work. We conclude with a discussion of related work and a summary of the chapter's contributions.

4.2 Overview of the Reference Architecture

The reference architecture addresses the *engine* software for Océ document processing systems (copiers). A copier engine is the part of the system that handles either the scanning or the printing of documents. Figure 4.1 illustrates the workings of a copier. A scanner engine extracts an image from the original sheet, whereas a printer engine reproduces the image data on blank sheets. The reference architecture describes an abstract engine that can potentially be used for any Océ copier.

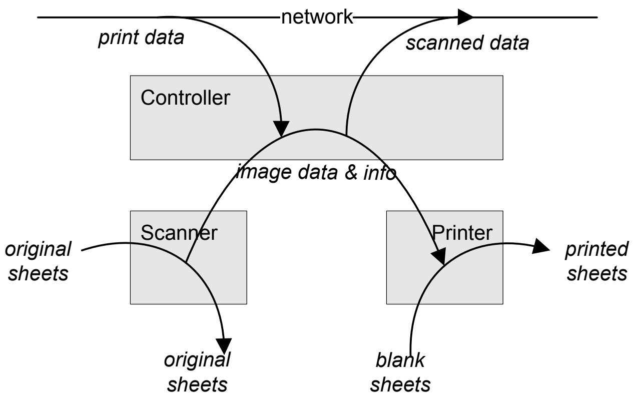


Figure 4.1: Main flows in a copier.

4.2.1 Business Drivers

Océ's reference architecture serves several purposes, of which the most important are:

Knowledge base It provides common terminology for software architects that is applicable to several products. The shared terminology together with the regular meetings dedicated to development of the reference architecture enable architects to share experiences more efficiently.

Starting point Its documentation can be used by new projects as a starting point for Océ's iterative development process. This greatly reduces the effort required for designing an engine architecture for a new product.

Reuse It describes the generic structure and behaviour of the engine software components. This makes integrating existing software components that are compliant to the reference architecture easier, and thus increases the reuse potential of those components. This not only includes binary components, but also designs, requirements and other software artefacts.

In fact the three points above are all related to reuse (i.e., of knowledge, documentation, and other software products). Therefore, the reference architecture should make it possible to eventually speed up the development (fast prototyping) and maintenance of products significantly.

4.2.2 Reference Architecture

The reference architecture defines the fundamental elements, relations between these elements, and properties of other, product-specific elements of Océ's copier engine software. It is used to derive a software architecture for engines incorporated in a specific series of Océ printers. From this software architecture, individual engines can be configured to be integrated in Océ's products. In this way the reference architecture defines a family of copier engines.

Deelstra et al. [2005] give a classification of product families with respect to level of reuse. We use this classification and the accompanying terminology to position the reference architecture. Four (ordered) levels are identified: 1) standardised infrastructure, 2) platform, 3) software product line, and 4) configurable product family. These levels denote to which extent the commonalities between related products in the product family are exploited. Océ's reference architecture can be positioned as a platform, since it provides reusable components that are developed by a separate reuse group (see Section 4.2.4). Furthermore, it defines a standardised infrastructure by prescribing how components should interact and what functional components should look like. Additionally, it offers a platform that realises common functionality, such as error handling and scheduling.

As all business drivers of the reference architecture are related to reuse, Océ is particularly interested in investigating whether it is possible and worthwhile to raise the current reuse level of the reference architecture to that of a product line. However, in order to qualify as a product-line architecture, it must define the functional variability between different engines.

Table 4.1: Views used in in the reference architecture's documentation

View \ Persp.	Static	Dynamic
Conceptual	System context, stakeholders, key requirements, external interfaces	Use cases, user visible states, configurations, variants
Logical	System components and dependencies, subsystem decomposition, persistent data, internal interfaces	Behaviour, component connection and disconnection, startup, key algorithms.
Physical	Files, directories, code, build rules	Threads, tasks, scheduling, interrupts.

4.2.3 Structure

The reference architecture is extensively documented using text illustrated with Unified Modeling Language[1] (UML) diagrams in more than 500 pages. The documentation is structured according to the *Architecture MetaModel* (AMM) developed by Atos Origin [Dinther et al., 2001]. AMM builds upon the Siemens four-views model [Soni et al., 1995] and Kruchten's 4+1 View Model [Kruchten, 1995]. It is organised around three types of views: *conceptual*, *logical*, and *physical* views. For each type of view, a static and a dynamic perspective is offered. This gives rise to six views, as illustrated in Table 4.1.

The documentation includes one *overview* document of approximately 50 pages, and a dozen documents describing the architecture for specific *concerns*, such as status control, software downloading, data persistence, and diagnostics. Each of these documents is organised according to AMM. The views are illustrated with diagrams expressed in UML-RT, a real-time extension of UML widely used at Océ [Dohmen and Somers, 2003]. In particular, many use cases are elaborated in sequence diagrams.

4.2.4 Usage

Currently the use of the reference architecture is voluntary. However, architects who want to use it for their project are supposed to first participate in the dedicated meetings for some months to get the same shared understanding of the reference architecture as the other participating ar-

[1]http://www.uml.org (June 2007)

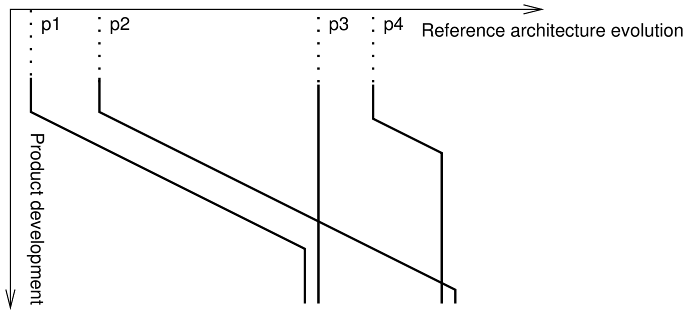


Figure 4.2: The reference architecture and derived projects.

chitects. This ensures that the reference architecture is more than a pile of documents. These meetings are very important as they provide a communication platform which is essential for meeting the initial objectives (Section 4.2.1).

In agreement with these objectives, there is a logical link between the reference architecture group and the group that develops reusable software components for the engine software. In the current situation, only the reusable components refer to the reference architecture's documentation, which means that this documentation itself does not show what components can be used to implement the different elements of the architecture.

The actual usage of the reference architecture leads to refinements and additions. Figure 4.2 depicts this interplay between usage and evolution. The horizontal line represents the evolution of the reference architecture. Each pi represents a project in which an engine is developed for a series of Océ copiers. A project can 'join' the reference architecture for some time, contribute to its development, and benefit from modifications made to it. This is indicated by the oblique lines for projects p1, p2, and p4. After a while, such projects may decide to 'leave' the reference architecture, and continue on their own using a fixed version (the lines become vertical). Other projects (p3) may decide to use a fixed version right from the start, extracting just whatever is necessary from that version of the reference architecture.

The reference architecture came into existence based on the documentation and experience of several previous projects. In fact, it was developed largely in parallel with one specific project. As such it can currently be understood as the common denominator of several product specific architectures.

As said using the reference architecture is voluntary and it is not yet known to all potential stakeholders. Therefore we can say that it is currently in an emerging phase. As such, besides confirmation that the reference architecture is suitable for its intended purpose, now and in the future, another result of its evaluation is the increased awareness of the potential benefits of the reference architecture with other development teams within Océ.

4.3 Evaluation Approach

The initial question that triggered this work was “How good is the reference architecture?” Additionally another important and related question was asked: “Does this reference architecture have a reason to exist?” The development team mainly wanted to get confirmation that the reference architecture is useful and that it is of good quality.

We first define what the terms ‘quality’ and ‘good’ mean in this context. As ‘good’ is always relative to particular requirements, the first step is to determine these requirements for the reference architecture, which were unknown since their definition was neglected during development.

As the reference architecture is intended to be used for several years and product generations, it is essential that it supports future changes to its environment and new product requirements. This is the main type of quality under consideration in the evaluation. Furthermore, in view of the fact that the objectives of this architecture as presented in Section 4.2 are centred around reuse, the impact that future changes will have on the reuse potential it offers, is essential. In the rest of this chapter we will use the term maintainability to refer to the type of quality required for a reference architecture described above.

Thus, the central question is: “How well is the reference architecture prepared for the future?” As this future is not always known at the time of evaluation, the selected method must explicitly address specification of possible extensions.

4.3.1 Selection of Evaluation Method

A literature overview of architecture evaluation methods [Dobrica and Niemelä, 2002] was used to select an appropriate approach to answer the central question above. Besides addressing maintainability as we described it in the previous paragraphs, Océ further required the method to be lightweight and well-documented. The method must have a low organisational impact because, as the reference architecture is still in an emerging phase, its evaluation must not affect other processes at Océ. Additionally, the method must be executable without additional training. This requires that a clear procedure for doing an evaluation based on the

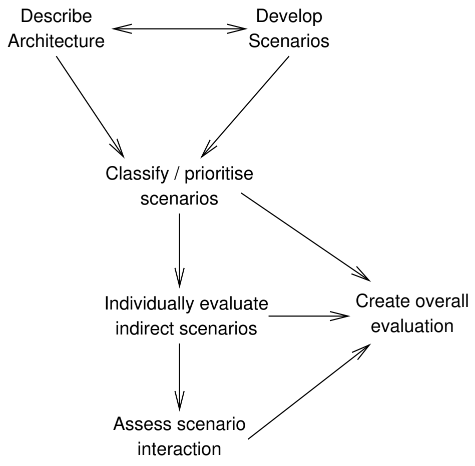

Describe Architecture
Develop Scenarios
Classify / prioritise scenarios
Individually evaluate indirect scenarios
Create overall evaluation
Assess scenario interaction

Figure 4.3: SAAM steps [Clements et al., 2002b].

selected method is available. These constraints imply the exclusion of many of the inventoried methods because these either focus on a different quality attribute or lack sufficient detail, e.g. many methods are defined and explained in only one published article.

The best-suited methods described in the inventory seem to be SAAM and its successor, the Architecture Tradeoff Analysis Method [Clements et al., 2002b] (ATAM). Both address maintainability and are extensively documented. Although ATAM is likely to produce more objective and accurate results, it also seems more difficult to apply for inexperienced assessors. The use of attribute-based architecture styles and their associated quality attribute characterisations for analysis of architectural decisions is not straightforward. Also the identification of sensitivity and trade-off points and the generation of a utility tree requires more effort and experience. Due to Océ's requirements with respect to the need for training (no need) and organisational impact (low) of the method, SAAM was selected.

In a SAAM evaluation, scenarios are developed to assess a software architecture's support for maintainability. The scenarios are used to express the required type of maintainability and thus SAAM can also be used to evaluate the type of maintainability we described previously. The developed scenarios represent possible future changes to the software system. An important aspect of SAAM is that it involves all stakeholders of a software architecture in a joint evaluation session, which results in a better appreciation and a more widely shared understanding of the software architecture.

Figure 4.3 shows the different phases of SAAM. A SAAM evaluation session starts with scenario development and description of the architecture. These are iterative activities. New scenarios can make it necessary to describe the architecture further, so that the architects can analyse them, while describing aspects of the architecture forces to think about possible scenarios addressing these aspects.

Next, the scenarios are prioritised and classified. Scenarios that can be realised without making changes to the current architecture are classified as *direct*. Scenarios that do require changes to the current architecture are classified as *indirect*. The indirect scenarios are evaluated for their impact. Furthermore, the scenario *interaction* is determined. Two scenarios interact when they require changes to the same architectural component. Information on scenario interaction is indicative of the quality of the decomposition.

Finally, the classification, prioritisation, analysis of the individual scenarios, and the scenario interaction are used to create an overall evaluation.

A SAAM evaluation session typically takes two days and involves an external evaluation team of three to four people. A session also involves system architects and other stakeholders. The type of stakeholders involved is very diverse: architects, developers, maintainers, integrators, managers, customers, end users, and so on.

4.3.2 Tailoring SAAM

SAAM has been selected as the evaluation method, yet it had to be tailored to Océ's situation. The current situation at Océ makes it necessary to modify SAAM for two reasons: 1) the organisational impact of SAAM and 2) the level of abstraction of the reference architecture.

In the situation of Océ the impact of gathering all potential stakeholders (as indicated in Table 4.2 on the next page), was considered too large. The main reason was that the stakeholders of a software architecture typically include some of the important members of an organisation that usually have very busy schedules. For the reference architecture this is especially true as it is the development group's ambition to make it a reference architecture that will impact development of many of Océ's copiers for years. Furthermore, the scope of a reference architecture is larger than that of a single-product architecture and therefore, next to a group of *direct* stakeholders a large group of *indirect* stakeholders (as indicated in Table 4.2 on the following page) exist, which makes the complete group of people with an interest in a reference architecture much larger.

The increased number of stakeholders made it impossible to find a date that suited all stakeholders and undesirable to take one or two full days of each stakeholders' time.

Besides the number of stakeholders the fact that we are studying a reference architecture also has an impact on the evaluation. It affects the level of abstraction; a reference architecture is more abstract than a single-product architecture.

The characteristics of the situation as found at Océ that we discussed above have several implications for the evaluation. Below we will discuss how these issues lead to modifications to the typical SAAM process as described by Kazman et al. [1996].

Table 4.2: Reference architecture stakeholders.

Stakeholder	Interest
Architects	Reference architecture architects
Users	Product architects
Management	Sponsors and decision makers
Potential users	Product architects not using the reference architecture
Reuse group	Provider of compliant components
Indirect	Stakeholders of products based on the reference architecture

The proposed tailored version is a distributed implementation of SAAM, called Distributed SAAM (DSAAM), that implements parts of the SAAM activities off-line, separately from the joint session. For instance, in preparation to the evaluation session, stakeholders are consulted individually. The joint SAAM session itself involves only participants fully aware of and well-informed on the reference architecture. The advantage of this approach is that the organisational impact is much smaller. Off-line consultation of individual stakeholders takes less time than a joint SAAM session. Additionally these consultations can be scheduled fitting the stakeholders' agenda's. Of course this approach increases the effort required by the assessors involved in these preparations. However, because we tried to minimise organisational impact, we aimed at reducing the required stakeholder effort.

An additional advantage is that smaller gatherings potentially induce less ambiguity, leading to a more efficient joint session. Therefore the actual DSAAM evaluation session lasts half a day instead of the usual two days. This further decreases the organisational impact of the evaluation.

4.4 Conducting the Evaluation

The evaluation consisted of roughly three phases. First, the joint DSAAM session had to be prepared. Second, the DSAAM evaluation session itself was executed. And finally an overall evaluation of the reference architecture was created. Three architects involved in the development of the reference architecture and two external observers participated in the joint session. One of the architects played the role of evaluation leader and prepared, chaired, and evaluated the joint session of DSAAM. For each SAAM step in Figure 4.3 on page 64, we explain below how it was included in the different phases of the DSAAM assessment.

4.4.1 Preparation

In preparation to the execution of the joint DSAAM session the available documentation (on the reference architecture and on SAAM) was distributed among the participants. The reference architecture's documentation was especially useful for the external observers as it explains the architecture and the applied architectural mechanisms. The documents on SAAM were only used by the evaluation leader.

The step 'develop scenarios' was carried out in two stages. During the preparation phase, the evaluation leader consulted stakeholders off-line. This resulted in an initial set of high-level scenarios representing possible futures from a stakeholder's perspective. The set of stakeholders included the sponsor of the reference architecture, members of the software reuse group, and hardware and domain experts. Unfortunately, the marketing and maintenance groups were not consulted, which limited the view on the road maps for Océ copier machines. The scenarios were related to either existing products or foreseen products. Whether the reference architecture was based on these products is irrelevant. The evaluation leader then added more detail to these scenarios according to a template for scenarios based on Bass et al. [2003].

4.4.2 Scenarios

In total sixteen scenarios were developed off-line. The majority of the scenarios aimed at reducing material costs, for example by sharing resources, using low-power designs, or offloading or re-mapping functionality. One scenario, for instance, aimed at moving functionality from the engine software to the main controller, another subsystem of a copier.

A second kind of scenarios was developed to reduce development costs. For instance, introduction of code generation for controllers of sensors and actuators based on mathematical models of those hardware devices. These scenarios were especially targeted at interactions which go beyond the domain level, such as communications with the mechatronics, testing, and manufacturing groups.

Finally, a minor source of scenarios involved an upgrade of the functionality, such as colour and wide-format printing. An example scenario is depicted in Table 4.3 in a format described by Bass et al. [2003].

Table 4.3: An example scenario.

Stimulus	Reduce power consumption by turning off parts of the copier machine during low-power mode
Response	Solve in engine specific projects
Source	Electronics department
Environment	Engine development time
Stimulated artefact	Reference architecture documentation
Response measure	Reuse percentage remains on same level

4.4.3 Execution

In the joint session each architect represented a product as a user of the reference architecture. Additionally, all architects played the role of assessor. As some of the participants had no experience in SAAM evaluations and to explain the steps of the DSAAM process, the session started with a brief introduction of the process. For the process observers also the role of the reference architecture in the organisation of Océ was explained.

The step ‘describe the architecture’ was largely omitted during the DSAAM session, since the DSAAM session only involved people that are well-informed with respect to the reference architecture and extensive documentation was already available.

The second part of the step ‘develop scenarios’ was done during the DSAAM session. This involved only architects of products on which the reference architecture was based. Apparently, the scenarios contributed by the stakeholders consulted prior to the joint session are representative for what may change in the future, as soliciting for extra scenarios gave no results. As such, the scenarios gathered and elaborated by the evaluation leader were used.

Scenarios were classified, prioritised, and evaluated as in SAAM, that is, during the session itself. The scenarios were classified and evaluated one by one, bypassing prioritisation (Figure 4.3 on page 64). Figure 4.4

gives an impression of the final result of the SAAM session. Scenarios were classified in directly and indirectly supported scenarios.

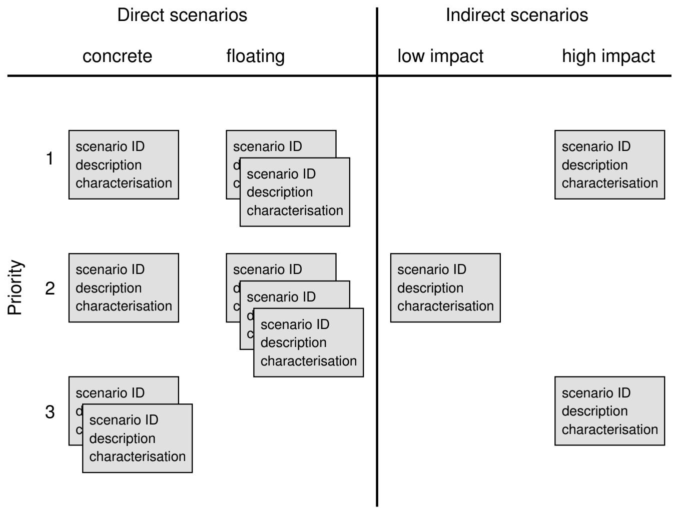


Figure 4.4: SAAM results

In general, first the impact of a scenario on a specific product was evaluated, and then its impact on the reference architecture. Classification and evaluation required a different attitude because we were evaluating a reference architecture instead of a product architecture. The difficulty lied in the fact that while scenarios are concrete, representing future functionality, or the quality of actual products, the reference architecture is abstract. The question:"What is the impact on the *reference* architecture?" needed to be answered consistently for all scenarios. Therefore we defined two types of direct scenarios:

1. Scenarios that are supported by the reference architecture as is and for which it provides concrete guidelines on how to realise them in product instantiations, and

2. Scenarios that can be realised by systems based on the reference architecture, but for which it does not (yet) provide detailed information on how to realize them (floating).

The class of floating scenarios calls for a cookbook with recipes that describe solutions for variation points in the reference architecture. Cookbook recipes describe how the reference architecture can be used to realize a specific (floating) scenario. For example, it might be necessary to describe what kind of components need to be defined or how some of the already defined components should cooperate to implement the desired behaviour. This information can be included in the reference architecture in a separate document without affecting the existing documentation. By realising scenarios this way the scope of reuse is extended and the reference architecture's classification moves from platform towards product line (see Section 4.2.2). An example of such a cookbook recipe was the description of how to realize sharing of hardware resources within an engine. The recipes were just new reference architecture documents. In fact, some of the existing documents already were such recipes, such as the document describing how function component should look like, without actually defining concrete function components. These documents had a different nature than the other documents that describe specific components of the reference architecture, like a scheduler. Figure 4.4 on the previous page shows that most of the scenarios fall in this category of direct scenarios.

The indirect scenarios were, as usual in SAAM, partitioned in two subsets: a subset with low impact and a subset with high impact. Overall this assessment session did not discover many design flaws. The architects spent most of their time on the single high impact, high priority scenario (multiple sheet paths).

The indirect scenario interaction was considered very briefly as only a few indirect scenarios were discovered. It was concluded that those did not interact.

4.4.4 Overall Evaluation

This final stage of the assessment involved the overall evaluation, which resulted in a set of strong and weak points. The set of strong points includes the aforementioned use of the reference architecture and its flexibility; most of the evaluated scenarios are directly supported.

The set of weak points includes a design flaw that prevents support for multiple sheet paths, which is required for duplex printing, for instance. Additionally, the reference architecture seemed incomplete as it missed several cookbook recipes. For example, recipes for sharing hardware resources and reusing engine parts amongst different engines in a single copier are currently not included. Another weakness was that variation points were not explicit in the documentation. Related to this issue is a missing structure for documenting an instantiation of the reference architecture, an engine generation, with respect to its documentation. It was not clear how conformance to and deviations from the reference architecture should be specified by projects that develop such an instantiation. Nevertheless, this is important for the maintainability of the reference architecture and its instantiations.

4.5 Discussion

Below we both discuss the implications of evaluating a reference architecture and using a distributed SAAM approach and we indicate where these lead to suggestions for future work and research questions.

4.5.1 Reference Architecture

Reuse Level In Section 4.2.2 we positioned the reference architecture as a platform. Furthermore its business drivers were all related to reuse (Section 4.2.1). Therefore the positioning raised two questions: is the positioning of the reference architecture correct for the current situation, and for the future? If correct, the current reuse positioning as a platform should be supported by links between the documentation of the reference architecture and the documentation of instantiated products. In view of the reuse positioning, we expect a considerable reduction in the effort of documenting a product instantiation compared to a single-product architecture approach. A prerequisite for this conjecture is that there must be a systematic way of documenting product instantiations with respect to the reference architecture. It is unclear whether such a systematic documentation process exists.

> **Research question** *Can we define and deploy a systematic process for documenting product architectures with respect to a reference architecture?*

In order to find out if product instances are documented with respect to their reference architecture in a systematic way reuse metrics are required [Poulin, 1997] to determine how much of the reference architecture documentation is reused in the product instance documentation. As an example of such a reuse metric, consider two indicative figures: the relative size and a normalised cohesion factor. The size factor calculates the lines of documentation of a product instantiation relative to the size of the reference architecture's documentation. The cohesion factor takes the number of references from the documentation of a concrete product to the reference architecture's documentation that handle variation points, normalised with the total number of references from the product instantiation documentation to the reference architecture's documentation. A low relative size and high cohesion factor indicate a high reuse factor and thus a systematic approach for reusing the reference architecture in product instantiations.

> **Future work** *Define a metric to position a reference architecture with respect to scope of reuse.*

With respect to the future reuse positioning of the reference architecture we would expect, looking at its reuse-oriented business drivers, that Océ aims to increase its reuse scope. This objective is supported by the identification of various direct floating scenarios, which will be implemented by the development team in a cookbook (Figure 4.4 on page 69). This implies that Océ indeed foresees that the reuse positioning of the reference architecture is raised from platform to software product line in the near future.

Updates Maintainability was the central quality aspect in the evaluation. One aspect of maintainability is the possibility to *update* the reference architecture with developments that take place in a product instantiation: during the oblique lines in Figure 4.2 on page 62. In order to successfully implement a proposed update two issues need to be considered: conformance and permissiveness.

Conformance is the extent to which the product architecture and reference architecture match. One must specify the update in agreement with the existing reference architecture. This is necessary, for example, to prevent specification of updates to components that do not exist at all in the reference architecture. The architecture of a product may undergo small changes during its development. Consequently, there may be a discrepancy between the product architecture and the reference architecture. The discrepancy may obstruct the transfer of architectural fragments, e.g., a cookbook recipe, from the reference architecture to the product architecture. But it may also obstruct the update of the reference architecture itself.

To detect these architectural discrepancies and suggest possible repairs, one could check the conformance by first using reverse engineering techniques to raise the level of abstraction of concrete product architectures and then compare the result with the reference architecture [Van Deursen et al., 2004]. Chapters 5 and ch:ewsa2005 investigate how to assess conformance of architecture specifications automatically.

> **Future work** *Develop a technique to measure the conformance of a product architecture with respect to the reference architecture on which it is based in order to assess the possibility to transfer fragments from a product architecture to the reference architecture.*

The bare fact that a product has an architecture that conforms with the reference architecture does not ensure by itself that a proposed update will be successful. The reference architecture also has to be permissive with respect to the update. The reference architecture must provide the flexibility to incorporate the proposed update. An update might violate some of the design decisions taken earlier; whether this is the case is in practice generally hard to assess. One reason for this is that design decisions are not completely documented. Most times only the structural effect of a design decision is documented. Documenting other aspects of design decisions, such as their rationale and effect with respect to (non)-functional requirements is often neglected.

> **Research question** *How can we document design decisions explicitly and how can we then use them to assess an architecture's permissiveness with respect to a proposed update?*

Use of Reference Architectures Besides its technical use as a starting point for product specific software architectures, the reference architecture served according to its objectives as a discussion platform for the software architects of different products. In that sense the reference architecture indeed is an efficient way to exchange experiences among product teams.

Another use of the reference architecture appeared during discussions in the DSAAM evaluation. It acts as a stable platform for negotiations amongst different domains: the mechatronics, manufacturing, and software reuse groups at Océ. By introducing a generic and more stable architecture for the engine software of Océ copiers the development group tries to prevent that software is automatically considered to be the means to solve problems during engine integration. As such defining an embedded software reference architecture helps creating a better balance between the different disciplines involved in engine development. This is a typical problem in the embedded software domain as was also observed in the inventory described in Chapter 3.

In the evaluation we conducted, the usage of the reference architecture was not addressed explicitly. Considering the specific use of *reference* architectures described above, it seems useful to do so, especially in the case of embedded systems.

> **Research question** *How can we include the usage of a reference architecture in an evaluation?*

4.5.2 Distributed SAAM

The main concern of scenario-based evaluation methods is whether the coverage and scope is broad enough to be conclusive about the findings of the evaluation. SAAM overcomes this by organising a general two-day gathering, which is moderated by experienced assessors. In DSAAM we had to take alternative measures.

In view of the two questions above, the number of *direct* stakeholders of the reference architecture is limited (see Table 4.2 on page 66), although many *indirect* stakeholders can be identified. These two groups of stakeholders seem to have different interests.

Raising the scope of reuse of the reference architecture directly concerns the architects of compatible products as its users and architects. It implies that the reference architecture not only should identify variation points but also explicitly give alternatives. The cookbook of the previous section provides these.

Scenarios that describe future development of existing and foreseen products are the concern of the stakeholders of those products. The development of these scenarios is the responsibility of these stakeholders, which are not necessarily also direct stakeholders of the reference architecture. On the other hand the resulting scenarios are input to DSAAM session, thus indirectly they are.

One measure we took to include indirect stakeholders in the evaluation was to split the process of developing the set of scenarios in two stages: an off-line stage with the indirect stakeholders, and a DSAAM stage with the direct stakeholders. The scenarios provided by the indirect stakeholders were product specific. Evaluating the impact on the *reference* architecture was not their concern, but that of the *direct* stakeholders. Furthermore the direct stakeholders are the only ones capable of doing so. Therefore because only the indirect stakeholders were excluded from the joint session, the scope of the DSAAM session was not affected by the lack of stakeholder interaction during evaluation.

However, this also prevented indirect stakeholders to interfere or interact during scenario prioritisation. During the DSAAM session, the architects concentrated on the most likely scenarios, from the perspective of an architect. Although scenarios were prioritised with respect to their impact, there was no clear rationale for this ranking. Hence DSAAM's scope was still at risk due to the possibility of a wrong scenario prioritisation.

In order to validate DSAAM's scope we recommend to organise indirect stakeholder involvement after the joint session. During this feedback phase stakeholders might be consulted in small sessions or individual interviews, in the same way as we did in preparation to the session. This time the indirect stakeholders can comment on the scenarios prioritisation and verify whether the evaluation covered all relevant aspects of the architecture. This preserves the small impact on the organisation offered by DSAAM. During the feedback phase, indirect stakeholders may conclude that some likely scenarios have not been evaluated thoroughly enough. Thus the feedback phase may yield newly developed scenarios. This new set of scenarios has to be evaluated in a new DSAAM session.

Future work *Extend DSAAM with an off-line feedback phase after the joint session for indirect stakeholders.*

Use of Documentation During the assessment we were somewhat surprised that the actual documentation of the reference architecture was not used at all during the session. This means that the architecture assessed is the one that is in the team members' heads, and not the *documented* architecture. The corresponding risk is that the team may have different architectures in their minds, that the documented architecture is inadequate, and that architects not participating may have different perspectives. Thus we have:

> **Research question** *How can we involve the architecture as documented explicitly in the assessment process?*

Solution directions will require explicit, analysable representations of both the architecture and the scenarios used in the assessment. An interesting research topic is whether information retrieval techniques can be used to analyse the relationship between these two representations.

4.6 Related Work

An overview of SAAM and ATAM, as well as references to many other methods for evaluating software architectures can be found in the book by Clements et al. [2002b].

Gallagher [2000] discusses the application of ATAM to a reference architecture. Unfortunately, he hardly discusses any issues specific to the evaluation of reference architectures (such as the different role of scenarios). The reference architecture is more or less evaluated as a single-product software architecture with specific business drivers.

Since the boundary between product line architectures and reference architectures is not always distinct (Section 4.2), another area of relevant related work is the field of product line evaluation. Lutz and Gannod [2003] discuss the architectural analysis of a product line architecture. The authors present a three-phased approach consisting of architecture recovery, scenario-based assessment, and model checking of safety-critical behaviour. Here a software architecture needed to be recovered from an existing product, which is then evaluated in order to see whether this type of product is amenable to a product-line-development approach.

Of particular interest are evaluation methods focusing on *maintainability*. The architecture-level modifiability analysis [Bengtsson et al., 2004] (ALMA) method integrates a number of different scenario-based approaches for assessing architecture maintainability.

4.7 Conclusion

In this chapter we reported the evaluation of an embedded software reference architecture using a tailored SAAM-based approach. The objective of the assessment was to assess the maintainability of the architecture. Maintainability involved two aspects, raising the scope of reuse from a platform to a product line and facilitating anticipated extensions of derived products and future products.

The evaluation of the reference architecture was based on a distributed SAAM (DSAAM) method, involving three phases: a preparation phase in which indirect stakeholders are consulted individually to collect scenarios, a joint evaluation session with only architects and observers, and an evaluation phase.

Assessing a reference architecture is different from assessing a product architecture. In an ordinary SAAM session, evaluated scenarios are categorised in directly and indirectly supported scenarios. We subdivided the set of directly supported scenarios into those with evidence of being supported by the reference architecture and those without evidence. The latter class typically consist of scenarios for which solutions are available in one of the products, but these have not been documented yet. In the DSAAM session we defined a cookbook to cover these scenarios.

The experience provided valuable insights for industry as well as for academia. In retrospect we argued that DSAAM is a suitable approach for the given situation, assessing the maintainability of a maturing reference architecture. Both the coverage of DSAAM and the quality of its conclusions are tenable. Note that reference and product-line architectures enable efficient reuse, a key business driver in many organisations. The concepts on which this type of architectures are based are maturing. Therefore it is expected that more and more companies will adopt a product-line approach, possibly involving reference architectures.

Océ gained insight in the positioning and status of the reference architecture in their organisation, its current position, and its future position. Océ also gained confidence in its maintainability.

We gained insight in the process of assessing a reference architecture. For instance, scenarios are typically evaluated based on a product instance and the results are abstracted to the reference architecture. This evokes all kinds of questions related to topics such as conformance checking and documenting design decisions, as discussed in Section 4.5.

Chapter 5

Model-Driven Consistency Checking of Behavioural Specifications[1]

For the development of software intensive systems different types of behavioural specifications are used. Although such specifications should be consistent with respect to each other, this is not always the case in practice. Maintainability problems are the result. In this chapter we propose a technique for assessing the consistency between two types behavioural specifications: scenarios and state machines. The technique is based on the generation of state machines from scenarios. We specify the required mapping using model transformations. The use of technologies related to the Model Driven Architecture enables easy integration with widely adopted (UML) tools. We applied our technique to assess the consistency between the behavioural specifications for the embedded software of copiers developed by Océ. Finally, we evaluate the approach and discuss its generalisability and wider applicability.

5.1 Introduction

System understanding is a prerequisite for modifying a software intensive system [Lehman and Belady, 1985]. As such the (typical) absence of up-to-date design documentation hampers successful software maintenance and evolution. In this chapter we address this problem for the documentation of a system's behaviour. We focus on ensuring the consistency between two types of behavioural specifications: interaction-based and state-based

[1]This chapter was published earlier as: Graaf, Bas and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 115–126. IEEE Computer Society, 2007a

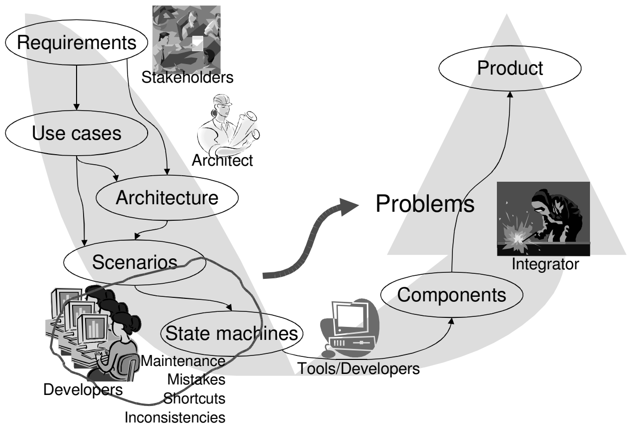


Figure 5.1: Typical development process

behavioural models. The use of such specifications is illustrated by the development process depicted in Figure 5.1. It is based on the well-known V-model [Bröhl and Dröschel, 1995] and the starting point of our research.

On the left branch of the 'V' analysis activities take place. Based on Requirements, the high-level Architecture is defined. This architecture identifies the main components of the system and assigns responsibilities. In parallel requirements are made more concrete by Use cases that specify typical interactions a user may have with the system. One distinctive property of use cases is that the system is considered to be a *black box* [Jacobson, 1992]. These use cases are the first interaction-based behavioural models.

Based on the use cases a set of Scenarios is defined that specifies the interactions of the system's components in terms of exchanged messages. Typically, every use case results in one (normal behaviour) or more (including exceptional behaviour) scenarios. These scenarios are also interaction-based behavioural models, but now the system is considered to be a *white-box*; they show the interactions between the components defined by the architecture.

Eventually, the architecture's components need to be implemented. This requires a complete behavioural specification. Scenarios are, however, not intended to provide such a specification for an individual component. First, the specification of a component's behaviour is scattered across multiple scenarios. Second, they are usually only defined for the components' most typical and important behaviours. Therefore, a complete state-based behavioural model, a State machine, is created for each component based on the set of scenarios. This state machine is used to implement or generate the component. Finally, on the right-hand side of the 'V', the different components are integrated into a complete product.

Such a software development process, where state-based component design is based on the specification of a set of use cases, is advocated by many component-based, object-oriented, and real-time software development methods [D'Souza and Wills, 1998; Kruchten, 1998; Jacobson et al., 1999; Selic et al., 1994]. As such, many software development organisations deploy similar development processes.

As software evolves it is often the case that changes are made to 'downstream' software development artefacts (such as designs) without propagating the changes to the corresponding 'upstream' software development artefacts (such as requirements). This can be the result of change requests, but also of design flaws that are only discovered on a more detailed level. Other inconsistencies are simply introduced by misinterpretations of 'upstream' development artefacts.

In this chapter we focus on inconsistencies between interaction-based behavioural models and state-based behavioural models. Inconsistencies between these models can be particularly important because they decompose behaviour along different dimensions. Interaction-based models are decomposed according to the different use cases, that is, they are *requirements*-driven. State-based models, on the other hand, are decomposed according to the different components that were identified during architecture design, that is, they are *architecture*-driven. This makes it hard to discover inconsistencies [Amyot and Eberlein, 2003; Bontemps et al., 2005]. Furthermore, when different development groups are responsible for the development of the different architectural components, and these groups individually resolve inconsistencies in different ways, this may obviously lead to problems during integration and maintenance.

In industrial practice behavioural models are often specified using the Unified Modeling Language[1] (UML). Moreover, tools are available that, based on UML, are capable of generating source code from such models. Considering such a model-based infrastructure, we believe it makes sense to view consistency checking of behavioural specifications as a model transformation problem. In this chapter we investigate what the advantages and disadvantages are of using model transformation technology to discover inconsistencies between interaction-based and state-based behavioural models. Furthermore, we aim to minimise the impact of our approach on existing development processes, for instance, in terms of the languages and tools used.

In Section 5.2 we introduce the industrial case that motivated this chapter: an embedded software control component developed by Océ, a large

[1]http://www.uml.org (June 2007)

copier manufacturer. At Océ an important copier subsystem is developed using a process corresponding to Figure 5.1 on page 78. Moreover, the components for this subsystem are generated from state machine models. As such, debugging, for instance, is performed on the level of state machines. As a result inconsistencies between scenarios and state machines become even more likely, making it a concern for Océ. Other work on the relation between scenarios and state machines is discussed in Section 5.3. The enabling technologies for our approach, as well as, the relevant part of the underlying UML specification, and our process for consistency checking are discussed in Section 5.4. In Section 5.5 we customise an existing mapping between scenarios and state machines based on Whittle and Schumann [2000] for specification as model transformations and consistency checking.

The application of our approach to the Océ case requires that the scenarios as found in Océ's architecture documentation are normalised into a form suited for the model transformations of our approach. After normalisation and application of our approach we identified several inconsistencies in the behavioural specifications that could lead to integration and maintenance problems. This is discussed in Section 5.6. Finally, we reflect on our approach in Section 5.7 and conclude with an overview of the contributions of this chapter and opportunities for future work in Section 5.8.

5.2 Running Example

Our motivation for investigating the consistency between interaction- and state-based behavioural models comes from a product-line architecture for software in copiers developed by Océ. We use this architecture as our running example and case study, and for that reason briefly explain it first.

At Océ a reference architecture for copier *engines* is developed. In a copier both the scanning and printing subsystems are referred to as an engine. The reference architecture describes an abstract engine that can be instantiated for (potentially) any Océ copier.

As a running example we use one of the reference architecture's components: the Engine Status Manager (ESM). This component is responsible for handling status requests and status updates in the engine. ESM and the other main components of the reference architecture are depicted in Figure 5.2 .

In a copier engine ESM communicates with two types of components: status control Clients, and Functions. Clients request engine state transitions. Requests by the external status control client (Controller) are translated by the EAI (Engine Adapter Interface) component. To perform status requests of Clients, ESM controls the status of individual Function components. Functions, in turn, recursively control the status of their composing Functions.

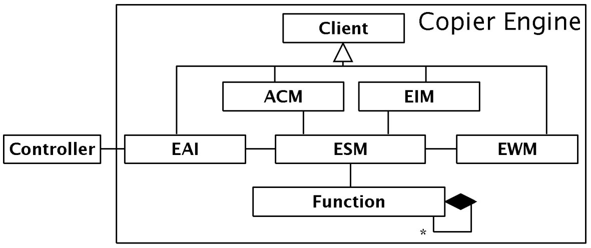


Figure 5.2: Architecture for copier engines

For the development of ESM and other components, a process is used similar to the process outlined in Section 5.1. For this Océ relies on a model-driven approach based on UML [Dohmen and Somers, 2003]. Architects specify use case realisations using UML sequence diagrams. Based on these diagrams, for every component a UML statechart diagram is created. Using special tooling[1], the source code for the engine components (e.g., ESM) is largely generated based on those statechart diagrams. For Océ's developers these statechart diagrams actually *are* the implementation.

One of the reasons for introducing a (automated) model-driven development approach was to overcome consistency problems with respect to state machine models and source code [Dohmen and Somers, 2003]. By automatically generating source code from state machines this problem is effectively moved 'upwards' to the consistency between scenarios and state machines.

For ESM, each use case addresses a specific engine state transition. A use case is accompanied by a UML sequence diagram. As an example, consider the diagram in Figure 5.7(a) on page 96. It depicts the interaction that occurs when a copier engine is requested to go to standby, while it is running. At Océ these sequence diagrams are purely used for communication purposes, rather than as input for automatic processing (e.g., model transformations, or code generation). Because of this, they are not always complete and precise. Furthermore, proprietary (non-UML) constructs are used. As an example, in these sequence diagrams the lifeline of the ESM component is decorated with the name of its (high-level) state at that point of the interaction.

To ensure successful evolution and maintenance of the reference architecture and the components it defines, a means to assess the consistency between the involved behavioural specifications is essential. It is this challenge we address in this chapter.

[1]IBM Rational Rose Technical Developer - http://www.ibm.com/software/awdtools/developer/technical (June 2007)

5.3 Related Work

Several formal approaches have been proposed that address problems similar to ours. Lam and Padget [2003] translate UML statecharts into π-calculus to determine behavioural equivalence using bisimulation. Schäfer et al. [2001] present a tool that uses model checking to verify state machines against collaboration diagrams. The use of such tools and approaches requires complete, precise and integrated interaction- and state-based behavioural models. This implies, for instance, that sending and reception of messages in scenarios are explicitly linked to events and effects in state machines. In our case, for the sequence diagrams, this is problematic. They are created early in the development process and not intended to be complete or precise.

To take this into account, we generate a state machine from a set of input scenarios, that, subsequently, is compared to the state machine that was created by the developers.

Many approaches have been defined for synthesis of state-based models from scenario-based models. Amyot and Eberlein [2003], and Liang et al. [2006] both evaluate over twenty of them. Evaluation criteria include languages, means to define scenario relationships and state model type. Our industrial case gives us the requirements with respect to these criteria for a synthesis approach.

Instead of using a more powerful scenario language such as live sequence charts [Damm and Harel, 2001], we limit ourselves to UML sequence diagrams augmented with decorations, as dictated by our industrial case study. The decorations with state information can be interpreted as conditions from which inter-scenario relationships can be derived. Finally, with respect to state model type, we consider approaches that result in state models for individual components (instead of global state models). Considering Liang et al. [2006] one approach best meets these requirements, namely the one proposed by Whittle and Schumann [2000].

Whittle and Schumann [2000] present an algorithm to map UML sequence diagrams to UML statecharts. In this mapping the messages in a scenario are first annotated with pre- and postconditions on state variables, referred to as a domain theory. The mapping is based on the assumption that a message only affects a state variable if its pre- or postcondition explicitly specifies it does; the domain theory does not need to be complete. Thus, this so-called frame axiom[1], together with the pre- and postconditions, results in a pair of state vectors for each message (before and after).

[1]The name derives from a common technique used by animated cartoon makers called framing where the currently moving parts of the cartoon are superimposed on the 'frame', which depicts the background of the scene, which does not change (http://en.wikipedia.org/wiki/Frame_problem (June 2007)).

For every scenario it is checked whether the message ordering is consistent with the domain theory. If not, either one can be reconsidered. Then, for each scenario a 'flat' state machine is generated for every component. Messages towards a component result in an event that triggers a transition; messages directed away from a component result in an action that is executed upon a transition. Loops are identified by detecting states that have unifiable state vectors. Two states vectors are unifiable if they do not specify different values for the same state variable. Subsequently, the 'flat' state machines generated for a component from different scenarios are merged by merging similar states. Two states are similar if their state vector is identical and they have at least one incoming transition with the same label. Hierarchy is added to the resulting statecharts by a user provided subset and (partial) ordering of the state variables.

Van der Aalst et al. [2004] present an approach for the discovery of (business) process models from event logs. Instead of state models, they use Petri nets as process models. Where they only rely on event (message) sequence to merge different workflow instances (scenarios), we rely on state variables as well.

Most work in this area focuses on the synthesis algorithm, whereas the integration in industrial practice remains implicit. In fact, many of the approaches are not supported by a tool or validated in industrial practice. Their application in practice only becomes realistic when they integrate with existing tools and standards used in industry. Therefore, we focus in this chapter on *UML* sequence diagrams as a notation for scenarios, and on *UML* state machines.

5.4 Model-Driven Consistency Checking

In this section we outline our approach for consistency checking of behavioural specifications, but, first, we introduce the technologies that enable our model-driven approach and the underlying structure of the involved behavioural models.

5.4.1 Enabling Technologies

Our approach takes advantage of the standards that are widely used in industry, such as UML and XML Metadata Interchange[1] (XMI), enabling easy integration with the tools used in industrial practice. XMI provides a means to serialise UML models to be manipulated, for instance, using Extensible Stylesheet Language Transformations[2] (XSLT). However, the XMI format

[1]http://www.omg.org/mda/specs.htm#XMI (June 2007)
[2]http://www.w3.org/TR/xslt (June 2007)

is very verbose, making it a tedious and error prone task to develop such transformations [Van Dijk et al., 2005].

Model Driven Architecture[1] (MDA) developed by the Object Management Group[2] (OMG) offers, among others, a solution to this problem. MDA is OMG's incarnation of model-driven engineering (MDE). With MDE, software development largely consists of a series of model transformations mapping a source to a target model. Essential to MDE are models, their associated metamodels, and model transformations. In the case of MDA, metamodels are defined using the MetaObject Facility[3] (MOF). The UML metamodel is only one example of such metamodels. Finally, model transformation languages are used to define transformations.

We use the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL)[4] to specify and implement the mapping between scenarios and state machines. ATL is used to develop model transformations that are executed by a transformation engine. With ATL, transformations are defined in transformation modules that consist of transformation rules and helper operations. The transformation rules match model elements in a source model and create elements in a target model. To this end the rules define constraints on metamodel elements in a syntax similar to that of the Object Constraint Language[5] (OCL). A helper is defined in the context of a metamodel element, to which it effectively adds a feature. Helpers can be used in rules, and optionally take parameters.

The ATL transformation engine can be used with XMI serialisations of models and metamodels defined using the MOF. For the sequence diagrams and state machines in this chapter we used the MOF-UML metamodel available from the OMG [OMG, 2007a]. To create the associated models, a UML modelling tool supporting XMI export can be used (we used Poseidon for UML[6] for this purpose).

Once the source model and metamodel, target metamodel, and transformation module are defined and located, the ATL transformation engine generates the target model in its serialised form, which, in turn, can be imported into a UML modelling tool for visualisation, or can serve as source model for another (model) transformation.

[1]http://www.omg.org/mda (June 2007)
[2]http://www.omg.org (June 2007)
[3]http://www.omg.org/mof (June 2007)
[4]For a more detailed introduction to ATL, please refer to Section 2.3.3.
[5]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)
[6]http://www.gentleware.com (June 2007)

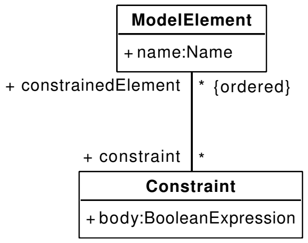


Figure 5.3: Constraints

5.4.2 Behavioural Modelling

For the creation of interaction-based and state-based behavioural models we use UML sequence and statechart diagrams. The underlying structure of these diagrams is described by the Collaborations and State Machines subpackages of the UML metamodel. Because our transformation rules are defined on the metamodel level, we introduce these package briefly. Although we discuss only simplified versions of these packages, the implementation of our technique and our case study are based on the complete UML metamodel (version 1.4 [OMG, 2007a]).

In general the UML metamodel allows every model element to be associated with an ordered set of constraints as can be seen from Figure 5.3. Note that every model element in UML is a specialisation of ModelElement. We use this to add pre- and postcondition to messages and state invariants to states. To distinguish between preconditions, postconditions, and other constraints that might be used in the model we use stereotypes.

Source: Collaborations The Collaboration package and some other UML elements are depicted in Figure 5.4 on the following page. In the context of a Collaboration the communication patterns performed by Objects are represented by a set of Messages that is partially ordered by the predecessor relation. For each message, sender and receiver Objects are specified. The cause of a Message is a CallAction (dispatchAction) that is associated with an Operation. In turn, this Operation is part of the Class that is the classifier of the Object that receives the Message. Finally, a Class optionally contains Attributes that have a type.

Target: State Machines Using the (target) metamodel in Figure 5.5 on the next page, UML state machines can be constructed that model behaviour as a traversal of a graph of state nodes interconnected by transition arcs.

A state node, or StateVertex, is the target or source of any number of Transitions and can be of different types. A State represents a situation in which

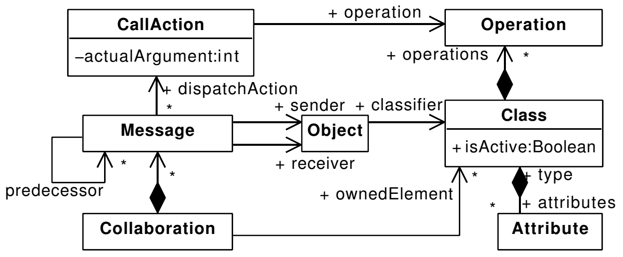


Figure 5.4: Collaborations (simplified)

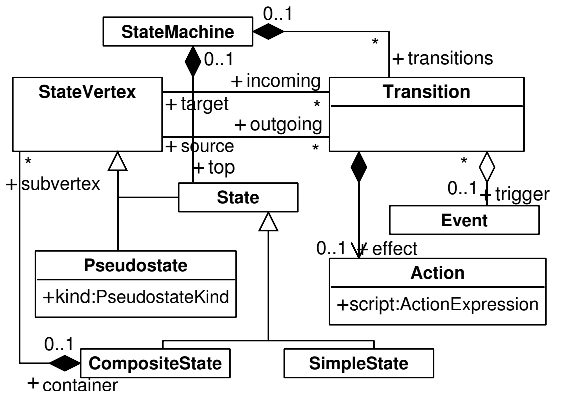


Figure 5.5: State machines

some invariants (over state variables) hold. The metamodel defines the following types of States. A CompositeState contains (owns) a number of sub-states (subvertex). A SimpleState is a State without any sub-states.

Next to state nodes that describe a distinct situation, the metamodel also offers a type of StateVertex to models transient nodes: Pseudostate. Only one Pseudostate type (PseudostateKind) is relevant for the state models in this chapter: the initial Pseudostate. An initial Pseudostate is the default node of a CompositeState. It has only one outgoing Transition leading to the default State of a CompositeState.

Nodes in a state machine are connected by Transitions that model the transition from one State (source) to another (target). A Transition is fired by a CallEvent (trigger). The effect of a Transition specifies an CallAction to be executed upon its firing. Finally, a StateMachine is defined in the context of a Class and consists of a set of Transitions and one top State that is a CompositeState (in the UML specification this is specified as an OCL well-formedness rule, which we do not show).

5.4.3 Consistency Checking Approach

As said, the set of scenarios is not expected to be complete or precise. For instance, when comparing the set of scenarios and the state machines created by the developers it is unclear whether a scenario specifies universal or existential behaviour [Damm and Harel, 2001]. However, if we are to generate a state machine for a set of scenarios we have to take a position with respect to the meaning of those scenarios. The generation of scenarios is based on the approach by Whittle and Schumann [2000]. To this end, we interpret Océ's scenarios in principle as universal: if the start condition of a scenario is satisfied the system behaves exactly as specified by that scenario. We consider the start condition of a scenario to be the first condition specified as decoration and occurrence of the first message. As such, the scenario in Figure 5.7(a) on page 96 specifies exactly what happens when ESM receives the message m_SetUnit(standby) while it is in state running. However, when during execution of a scenario the start condition of another scenario is satisfied, execution continues according to that scenario. For instance, in the case of Figure 5.7(a) on page 96, while ESM is stopping, execution could continue according to the scenario that performs the request of ESM going back to running while it was stopping.

In our approach we use model transformations for the generation of a state machine from a set of scenarios. The specification of those transformations is discussed in Section 5.5. To include all required information, the source model has to comply to a set of modelling conventions. When considering an arbitrary industrial case (e.g., Océ's reference architecture), the models used typically do not comply to those conventions. Therefore, we first require models to be normalised. This is discussed in Section 5.6.1.

Finally, the generated state machine is compared to the state machine that was already developed based on the same set of scenarios, the implementation state machine. Because the sequence diagrams are created early on in the development process, it is not expected that they are exactly covered by the state machines. Therefore, mismatches are expected between the generated and implementation state machine with respect to transition labels and order. This makes automating the comparison step particularly difficult. For now we manually compare the generated and implementation state machine and mainly focus on inconsistencies with respect to top-level states and transitions.

As such, we use three steps to check the consistency between behavioural specifications: normalise, transform, and compare. In the current approach only the transformation is automatic. Furthermore, the normalisation step is context-specific as it depends on the type of input models.

5.5 Generating State Machines

Given the source and target metamodels discussed in the previous section, we now describe how to instantiate source models, as well as the mapping between source and target models, expressed as ATL model transformations. We published all (executable) ATL transformations that we implemented, as well as (normalised) source and target (meta)models for the ATM example of Whittle and Schumann [2000] in the ATL transformations repository[1].

5.5.1 Instantiating a Source Model

Our approach based on model transformations and UML requires that all necessary information is encoded in a UML model. Whittle and Schumann [2000] requires the following information for its mapping: scenarios, a domain theory, a set of state variables, and an ordered subset of that set.

The set of scenarios is specified as sequence diagrams. The types of the interacting Objects (components) are specified in a class model. The Class that corresponds to the component of interest is marked active. All Operations involved in the relevant scenarios are also specified. The pre- and postconditions of a domain theory are applied to these Operations as stereotyped Constraints. These Constraints have the form `state variable = value`. We currently do not allow pre- and postconditions in the domain

[1]http://www.eclipse.org/gmt/atl/atlTransformations/UMLSD2STMD

theory that refer to formal parameters of the operations involved, as this would require interpretation of these conditions. If necessary, such constraints can be added directly to the Messages that specify an actual parameter in the sequence diagrams.

The active Class contains an Attribute for each state variable. The subset of state variables used for introducing hierarchy is encoded by setting the visibility of all state variables included in the subset to public and the others to private. Finally, the order of the state variable Attributes on the Class represents the prioritisation of state variables (the top one having the highest priority). This priority indicates the order in which the state variables are used to partition the set of states by assigning these states to CompositeStates according to the value assigned to the state variable.

5.5.2 Model Transformations

Our transformations generate a state machine for the component that is represented by the active Class in the source model. A scenario specifies one particular path through the state machine for that component, on which it proceeds to the next state upon each communication. We refer to the state machine that only describes that path as a 'flat' state machine.

We tailored the approach in Whittle and Schumann [2000] (see Section 5.3) to account for the type of input in the Océ case, for our model-driven strategy, and for our goal: consistency checking. For this reason we introduce fewer abstractions, making detecting and resolving inconsistencies more convenient. Our mapping consists of four separate steps: 1) apply the domain theory, 2) generate flat state machines, 3) merge flat state machines, and 4) introduce hierarchy into the merged state machine.

We formalised our mapping from scenarios to state machines as four ATL model transformations that correspond to the four steps of our mapping. Every consecutive transformation uses the target model of the previous transformation as its source model.

Together, these transformations are specified in less than 700 lines of ATL code. Before these transformations can be applied to the Océ case, a normalisation step is required, which is discussed in Section 5.6.1.

Apply Domain Theory This step is specific to our approach. Unlike Whittle and Schumann [2000], but in accordance with UML, we distinguish between pre- and postconditions on the Operations of a Class and those on the CallActions associated with Messages in a sequence diagram. This has two advantages. First, it allows for simple pre- and postconditions to be specified only once (i.e., on a Class' Operations). Second, it circumvents the

```
rule ConstrainedCallAction {
 from ca_in:UML!CallAction
 to ca_out:UML!CallAction(
   operation <- ca_in.operation,
   constraint <- ca_in.operation.constraint->union(ca_in.constraint)
    )
}
```

Listing 5.1: Applying constraints to CallActions

need to evaluate conditions that refer to formal parameters of an Operation.

When we apply the domain theory to a set of scenarios, we simply attach the pre- and postconditions on the Operations of a Class to corresponding Messages to or from instances of that Class.

The ATL specification of this mapping is straightforward. The Constraints on an Operation are copied to Messages via their associated CallAction. To this end, the rule in Listing 5.1 matches all CallActions. For each it generates a CallAction, `ca_out`, in the target model and initialises its constraint feature with the constraints applied to the Operation associated with the matching CallAction. Note that the constraints are added to the constraints already applied to the matched CallAction (using the `union` operation).

The result is a set of sequence diagrams in which Constraints are applied to Messages based on the pre- and postconditions of a domain theory on Operations. See Figure 5.7(b) on page 96 for an example.

Sequence Diagrams → Flat State Machines The next step of our approach is to generate a flat state machine for every scenario in which the component of interest plays a role. In this step we map every communication to a Transition and a target State. The source State of this transition is the target State corresponding to the previous communication of the component in the scenario. As in the approach by Whittle and Schumann [2000] our strategy is as follows: if the involved communication was the receipt of a Message, we say the Transition was triggered by that Message. If the involved communication was the sending of a Message, we say the effect of the Transition was sending that Message.

Based on the pre- and postconditions applied to the Messages in the scenarios by the previous step, we calculate the state vector for each State. For this we 'propagate' pre- and postconditions through the sequence diagram by application of the frame axiom. The result is a set of flat StateMachines, in which state vectors are applied to States as a set of Constraints over state variables.

```
rule EffectTransition {
  from m:UML!Message (m.sender.isActive)
  to t_effect: UML!Transition(
    effect <- ca,
    target <- trgt,
    source <- ... ),
  ae:UML!ActionExpression ( ... ),
  ca:UML!CallAction ( ... ),
  trgt:UML!SimpleState (
    name <- ae.body+'_sent',
    constraint <- m.stateVector)
}
helper context UML!Message def: stateVector : Set(UML!Constraint) =
  let stateVectorPrev:Set(UML!Constraint) = ... in
  let pres:Set(UML!Constraint) = ... in
  let posts:Set(UML!Constraint) = ... in
  let sv:Set(UML!Constraint) =
thisModule.frame(stateVectorPrev,thisModule.frame(pres,posts)) in
    if thisModule.unifiable(stateVectorPrev,pres) then
      sv
    else
      sv.debug('INCONSISTENCY DETECTED!')
    endif
;
helper def: frame(frame:Set(UML!Constraint), framed:Set(UML!
 Constraint)): Set(UML!Constraint) =
  frame->iterate(c; cs:Set(UML!Constraint)=framed |
    if cs->exists(e|e.stateVariable=c.stateVariable) then
      cs
    else
      cs->including(c)
    endif)
;
```

Listing 5.2: Mapping Messages to (effect) Transitions

As an example, the `EffectTransition` rule in Listing 5.2 matches all Messages in the source model sent by the component of interest. The target pattern specifies that for each such Message (`m`) among others, a Transition (`t_effect`) and a SimpleState (`trgt`) are created in the target model. The effect and target features of the Transition element are simply initialised to the CallAction (`ca`) and SimpleState created in the same rule. The source of the Transition is initialised to the target of the Transition that corresponds to the previous Message (not shown).

The constraint feature of the generated SimpleState element is initialised to the set of constraints (state invariants) that hold after the Message that matched the rule. This is determined by the `stateVector` helper. For this it applies the frame axiom (specified in the `frame` helper)

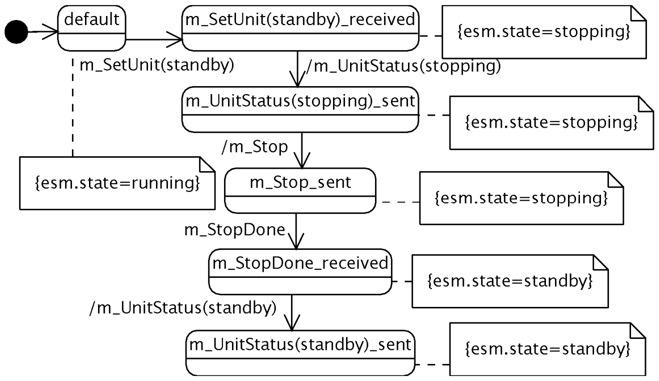


Figure 5.6: Flat state machine

subsequently to the postconditions of the current Message (`'posts'`), the preconditions of the current Message (`pres`), and the state vector after the previous Message (`stateVectorPrev`). As such conditions propagate in 'forward' direction (i.e., downwards in a sequence diagram).

Additionally the `stateVector` helper notifies the user if an inconsistency is detected between the state vector after the previous Message and the preconditions for the current Message (these sets of Constraints should be unifiable).

The `frame` helper simply iterates over the Constraints in the `frame` argument and adds every constraint involving a state variable that is not referred to in `framed` to that set.

Unlike Whittle and Schumann [2000], we do not apply unification of state vectors at this stage. The declarative style of our ATL specifications results in an infinite recursion: to complete a state vector we need to know whether it can be unified with other state vectors. To determine this we have to consider state vectors in 'forward' as well as in 'backward' direction. However, to determine the state vectors in 'forward' direction, we, in turn, have to consider state vectors in 'backward' direction because of the frame axiom strategy.

Application of this step yields a set of flat state machines for a component. As an example, consider Figure 5.6. It depicts the flat state machine corresponding to the sequence diagram in Figure 5.7(b) on page 96. Note that the example only involves a single state variable and that the names of the States are derived from the particular Message that was sent or received by the component.

```
rule MergedSimpleState {
  from s_in:UML!SimpleState (
    thisModule.mergedStates->includes(s_in))
  to s_out:UML!SimpleState(
    name<-s_in.name,
    constraint <- s_in.constraint)
}
helper def: mergedStates: Set(UML!StateVertex) =
  thisModule.allSimpleStates->union(thisModule.allPseudostates)
  ->iterate(s; mss:Set(UML!StateVertex)=Set{} |
    if mss->exists(e|(e.mergeable(s)) then
      mss
    else
      mss->including(s)
    endif)
;
helper context UML!StateVertex def: mergeable(s:UML!StateVertex):
 Boolean =
  thisModule.unifiable(self.constraint,s.constraint) and self.name=s.
   name
;
helper def: unifiable(cset1:Set(UML!Constraint),cset2:Set(UML!
 Constraint)): Boolean =
  let sharedSVs:Set(UML!Attribute) = cset1->collect(c|c.stateVariable
   )->select(a|cset2->collect(c|c.stateVariable)->includes(a)) in
    sharedSVs->forAll(a|cset1->select(c|c.stateVariable=a)=
      cset2->select(c|c.stateVariable=a))
;
```

Listing 5.3: Merging SimpleStates

Merging Flat State Machines In this step we merge the flat state machines. We merge every set of states with unifiable state vectors and at least one identical incoming transition (in terms of effect or trigger).

Merging of states is done by the rule and helpers in Listing 5.3. The rule matches all states selected by the `mergedStates` helper that iteratively selects one SimpleState from every group of equal SimpleStates in the source model. A call to the `mergeable` helper results in true when the receiving StateVertex and the parameter StateVertex 1) (`s`) are unifiable, and 2) have the same name (i.e., the incoming transitions had the same trigger or effect). The `unifiable` helper evaluates to true for two sets of Constraints that do not specify different values for the same state variable, meaning that the constraint that refers to a particular state variable that is also referred to in the other set, is actually included in that set.

Transitions are matched by another rule (not shown). To discard redundant Transitions, it only matches one Transition of the Transitions between any two sets of SimpleStates that are merged.

Introducing Hierarchy As suggested by Whittle and Schumann [2000], we use an ordered subset of the set of state variables to add hierarchy by means of CompositeStates. These state variables define a hierarchy of Composite-States. For instance, the state variable with the highest priority results in CompositeStates in the top level CompositeState: one for each value of that state variable's domain (provided that it occurs in one of the simple states' state invariants). For each of the resulting CompositeStates, the second-highest priority state variable, in turn, results in CompositeStates for each value of that state variable's domain that occurs in combination with the corresponding value of the higher-priority state variable.

The problem of specifying this mapping with ATL is that there is not always a matching source model element to create a CompositeState for. Therefore, we use an ATL *called* rule (`CompositeState`). A called rule is an imperative rule that is not matched by a source model element. Instead, it is explicitly called and can have parameters. The `CompositeState` called rule in Listing 5.4 creates a CompositeState for a given set of Constraints (`cset`). These Constraints (i.e., state invariants) are determined by the `compositeStateConstraintSetsAt` helper that takes a set of Constraints that represents the current CompositeState and determines the sets of Constraints that correspond to the CompositeStates at that level. For each of those sets a CompositeState is created. This called rule is used to initialise the `subvertex` feature in the rule that matches the top CompositeState of the merged StateMachine, as well as (recursively) in the `CompositeState` rule itself. The **do** clause in the `CompositeState` rule returns the created CompositeState.

5.6 Application to Océ

In this section we first explain what additional work has to be done to apply our approach to the Océ case. Subsequently we give an overview of the results obtained by application of our approach.

5.6.1 Source Model Normalisation

In the case of Océ, neither a domain theory, nor a set of state variables were available. To overcome this, we normalise Océ's sequence diagrams. In particular, we interpret the decorations on object lifelines as pre- and postconditions on a single state variable of a suitable enumeration type: `state`. The message preceding a state decoration apparently resulted in the component moving to the indicated state. Hence, we (manually) attach a corresponding postcondition to the message (e.g., `esm.state=starting`). A message succeeding a state decoration apparently requires the component

```
rule TopCompositeState {
  from cs_in:UML!CompositeState
  using {
    sm:UML!StateMachine=thisModule.allStateMachines->select(sm|sm.top
     =cs_in);
  }
  to cs_out:UML!CompositeState (
    name <- cs_in.name,
    subvertex <- sm.simpleStateStatesAt(Set{})
    ->union(sm.compositeStateConstraintSetsAt(Set{})
      ->collect(cs|thisModule.CompositeState(sm,cs))))
}
rule CompositeState (sm:UML!StateMachine, cset:Set(UML!Constraint))
 {
  to cs:UML!CompositeState(
    subvertex <- sm.simpleStateStatesAt(cset)->union(sm.
     compositeStateConstraintSeqsAt(cset)->collect(cs|thisModule.
     CompositeState(sm,cs))))
  do{cs;}
}
```

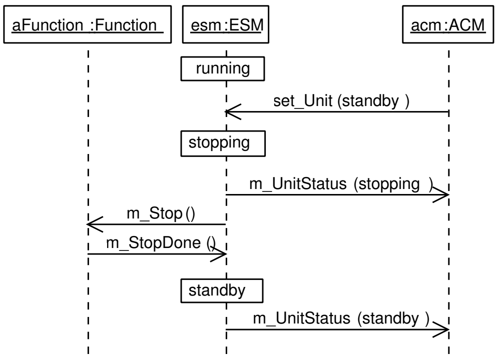
Listing 5.4: Adding hierarchy to state machine

to be in the indicated state. Hence, we attach a corresponding precondition to the message. Figure 5.7 on the next page shows an example. Finally, we add a (public) attribute, `state`, to the class corresponding to the ESM component.

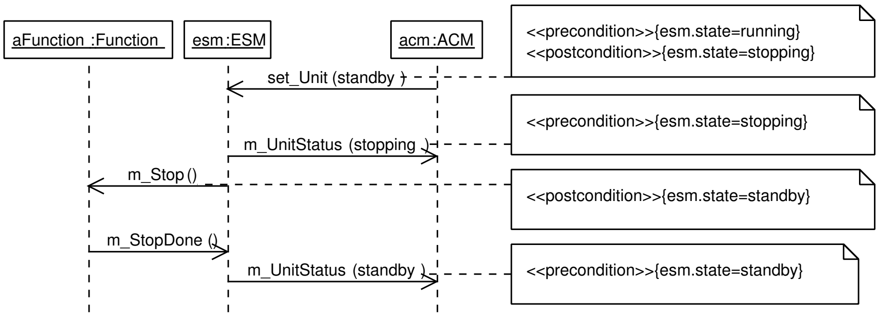
5.6.2 Results

A fragment of the result of application of the transformation steps to Océ's ESM component, is depicted in Figure 5.8 on page 97. The dashed line indicates the path through the state machine that is traversed when ESM is requested to go to standby while it is running. This path corresponds to the scenario depicted in Figure 5.7 on the next page.

We compared this *derived* state machine with the *implementation* state machine, from which Océ generates code. There are many inconsistencies with respect to low-level states and transitions. In the implementation state machine low-level states are not only decomposed further, the sequence of states and transitions is also different in many cases. This is not surprising considering the fact that the sequence diagrams of the source model from which we derived a state machine, constitute the first behavioural model that is created for the ESM component, while, in the implementation state machine, low-level transitions and states often correspond to a single method call in the generated code. If we restrict the



(a) Sequence diagram with decorated lifeline


aFunction :Function
esm :ESM
acm :ACM
<<precondition>>{esm.state=running}
<<postcondition>>{esm.state=stopping}
set_Unit (standby)
<<precondition>>{esm.state=stopping}
m_UnitStatus (stopping)
m_Stop ()
<<postcondition>>{esm.state=standby}
m_StopDone ()
m_UnitStatus (standby)
<<precondition>>{esm.state=standby}

(b) Normalised sequence diagram

Figure 5.7: Example scenario: request a copier engine to go to standby while it is running

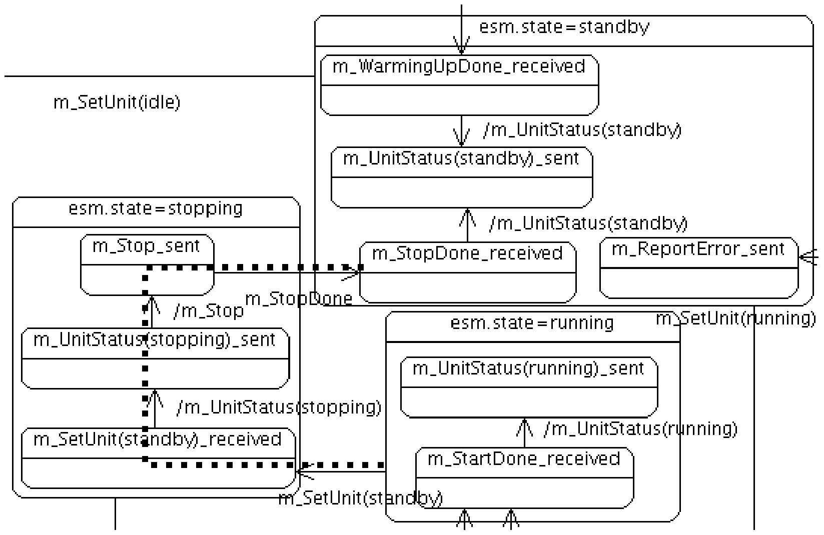


Figure 5.8: Merged state model of ESM (fragment)

comparison step to the top-level states, however, the implementation state machine largely conforms to the derived state machine. Although we cannot show the implementation state machine, we were able to make several other interesting observations:

- Several transitions between top-level composite states are missing in the derived state machine. This indicates that not all scenarios have been specified in a sequence diagram.

- Some top-level composite states in the derived state machine were modelled as low-level (sub) composite states in the implementation state machine. This merely indicates changes to the decomposition of states, and does not necessarily result in different behaviour.

- In the derived state machine, sometimes extra paths exist between two composite states. This indicates that specific sequences of events and actions that occur in different scenarios are not specified consistently. This was the case, for instance, when two versions of a scenario existed: one for normal behaviour, and one for exceptional behaviour. For two such versions the first interactions should typically be identical (until some exception occurs), but in practice this was not the case.

- The derived state machine contains a number of unconditional transitions that form a loop, resulting in non-deterministic behaviour. This had the same cause as the previous observation.

As a response to these observations Océ has two options. The first is to add missing use cases and scenarios, and to refactor alternative sequence diagrams to remove inconsistencies in event and action sequences. Here, care must be taken, as such modifications affect the state machines of other components that play a role in the involved scenarios as well. The alternative option is to only remove behavioural inconsistencies in the implementation state machine. This also requires careful analysis, since different development groups, responsible for different components, might do so differently, resulting in integration and maintainability problems.

5.7 Discussion

Generalisability of the approach *To a large extent our approach is generic.*

Although, the normalised source model in the Océ case only contains a single state variable, we also applied our transformation step to the ATM example in Whittle and Schumann [2000][1]. This example involves three state variables. By application of our approach (in both cases) we detected several inconsistencies.

We applied our approach successfully to both the ATM example of Whittle and Schumann [2000] and Océ's reference architecture. Our approach is generic with respect to input models that comply to the model conventions as outlined in Section 5.5.1. As such, we require a (manual) normalisation step that is context specific; it depends on the modelling conventions in use at a particular company.

Our modelling conventions are most restrictive with respect to the type of pre- and postconditions used in the domain theory. As we do not evaluate these conditions, we require them to be of the form `stateVariable=value`. In some cases the conditions for an Operation refer to a formal parameter. Our approach can still be applied if the Messages associated with that Operation in the sequence diagrams specify a corresponding actual parameter. Then, we (manually) apply the condition directly to the Message in the sequence diagram and substitute the formal parameter for the actual parameter. More complicated conditions require full interpretation of OCL expressions.

Of course, pre- and postconditions have to be available for our approach to produce more than only flat state machines. In the case of Océ, we de-

[1] Images of the (normalised) source model, as well as all (intermediate) target models for the ATM example can be downloaded from the ATL transformations repository (http://www.eclipse.org/gmt/atl/atlTransformations/UMLSD2STMD)

rived pre- and postconditions from decorations in the sequence diagrams. In general, pre- and postconditions are not always obvious from design documentation. In such situations these might have to be derived indirectly from documentation or reverse engineered from source code.

The introduction of pre- and postconditions effectively is a normalisation to the UML standard used by Océ and our tools (version 1.4 [OMG, 2007a]). For the current UML (version 2.1.1) this is not necessary, as such lifeline decorations became part of the specification (the corresponding metamodel element is called StateInvariant [OMG, 2007b, page 500]). To support this, only minor modifications to our ATL transformations are required.

Scalability of the approach *Our approach constitutes a first step towards fully automated consistency checking.*

In the Océ case study, the source model for the transformation step of our approach includes 10 sequence diagrams that contain 62 messages. The resulting integrated, hierarchical state machine, of which a fragment was depicted in Figure 5.8 on page 97, contains 23 transitions between 14 composite states containing in total 47 simple states.

Our approach is a first step to fully automated consistency checking of behavioural specifications. For now, we rely on manual inspection of the resulting state machine for actual evaluation of the consistency. As such, the scalability is currently not limited by the transformation steps (in the Océ case they each take less than 10 seconds), but by the comparison step. For cases were the number of states is limited and developers have knowledge on the system, this is a feasible approach. For ESM, which is a medium-sized component (approximately 10 KLOC), this turned out not to be a problem.

Fully automatic consistency checking could be done by relying on naming. An example of such an approach is discussed by Van Dijk et al. [2005]. It checks the consistency between the underlying XMI representations of UML models. However, in this case, the names of messages in the scenarios did not precisely correspond to the names of transition effects and events in the implementation state machine. Although they are easily matched by a human, this hampers full automation.

The use of graph matching techniques is another possibility of checking the consistency of two state machines. Also using such techniques the problem of matching node and edge labels remains.

Also for automatic approaches, however, the generation of a state machine from a set of scenarios, as discussed in this chapter, is likely to be a first step.

Applicability of the approach *Our approach can be applied to iteratively develop behavioural specifications.*

We generated a state machine with the purpose of checking the consistency between different behavioural specifications. However, our approach might have other applications as well. A generated state machine could also be used for other types of analyses, such as model checking or performance analysis.

Next to analysis purposes, our approach is particularly interesting for forward engineering, especially in the context of model-driven development approaches as in the case of Océ. Using our transformations based on UML, developers can easily generate different views on the behaviour of a software system or component. Furthermore, the generation not only provides insight in the consistency of the sequence diagrams with respect to each other, it also provides developers with a first candidate state machine that can be refined. As such, our technique can be applied iteratively to develop complete behavioural specifications of components: (1) specify the interactions of an initial set of use cases as scenarios, (2) generate a state machine, (3) refactor scenarios to remove inconsistencies in event and action sequences, and add missing scenarios, (4) go to step 2.

The main reason to choose for a model-driven approach based on UML for our consistency check, is the integration with Océ's development process. It circumvents the need to extract information from the MDA domain to another domain, such as the grammarware or the Extensible Markup Language[1] (XML) domain. Unfortunately, despite the availability of standards, currently available tools for (meta)modelling and transformations do not integrate well, hampering actual integration of our approach in practice. For a large part this is due to the abundance of possible combinations of XMI, UML, and MOF versions, as well as vendor specific implementations of those standards. Other problems occur due to different capabilities of modelling tools. As an example, we used Poseidon for UML to create source models because its metamodel is available from the developer's website. However, the UML models we generate do not contain layout information. Unfortunately, Poseidon is not capable of displaying UML models that do not contain layout information. As a consequence we had to use another tool for visualisation. From a large set of tools we tried, only Borland's Together[2] is capable of generating a layout for a UML model. However, the XMI representations used by this tool are not compatible with those generated by the ATL engine. As a workaround we developed a minimal XSLT transformation that maps the XMI 'flavour' generated by the ATL engine to that of Together. An alternative is to generate the layout information required by Poseidon using a model transformation.

[1]http://www.w3.org/XML (June 2007)

UML vs. MOF *The use of UML in a limited domain makes transformation definitions unnecessary complex*

The genericity and resulting complexity of the UML metamodel result in, sometimes, inconvenient navigation through source and target models to select a certain element. An open question is whether tree traversal strategies as present in Stratego[1] or the JJTraveller [Van Deursen and Visser, 2004] library could help to alleviate this navigation problem. Also, often relations are defined as $n : m$ while in a specific case $1 : 1$ would suffice. The result is that sets have to be converted to sequences of which the first element has to be selected. This is required very frequently, resulting in unnecessary complex ATL code.

In cases, where only limited parts of the UML metamodel are used, an alternative could be considered. Instead of using the UML metamodel, custom MOF-based metamodels could be used, for instance, for scenarios and state machines. These metamodels could be much simpler, resulting in simpler transformation definitions. The price to pay is that in order to establish a connection with actual UML models (e.g., as used by Océ), a mapping between such custom metamodels and the UML metamodel must be specified. In Chapter 8 we discuss how to make such a mapping for domain-specific modelling languages (DSMLs).

5.8 Conclusions

In this chapter we explored the use of model transformations to check the consistency between behavioural specifications. For this we presented an approach that consists of normalisation, transformation, and comparison steps. We consider the following to be the main contributions of this chapter:

- A specification of the mapping between scenarios and state machines using model transformations that is made available via the ATL transformations repository. An advantage of such a specification is that it can be executed by the ATL transformation engine. Furthermore, it is completely based on UML, making integration in industrial practice easier.

- Modelling conventions for encoding the information required for the transformation step in a single UML model. Additionally, as an example, we discussed the required normalisation step for Océ's reference architecture.

[1]http://www.stratego-language.org

- Validation of the proposed approach by application to an industrial system, demonstrating that even small industrial specifications (consisting of just 10 scenarios) contain inconsistencies, which are effectively identified by our approach.

Finally, the proposed approach can be applied for other purposes than consistency checking as well, such as forward engineering and early behavioural analysis based on the generated state machine.

Currently we are extending our work with additional case studies. Furthermore, we investigate the possibilities to do consistency checking automatically, again by the use of MDA model transformation technologies.

Chapter 6

Model-Driven Conformance Checking of Structural Specifications[1]

Nowadays, industry is confronted with rapidly evolving systems. In order to effectively reuse design artefacts such as requirements, architectural views and analysis results, as well as the code base, it is important to have a consistent overview in each phase of the development process. In this chapter we propose a conformance checking system based on views to evaluate the conformance of an implementation with respect to its architecture. We map our approach onto technology for model-driven engineering. An academic case study illustrates application of our framework.

6.1 Introduction

The current trend in embedded systems is product families rather than single products. Today's customers are appealed to products that have a sense of uniqueness, products that are compatible but slightly different than those of their friends. The answer from industry is to develop flexible product lines. The extent to which the evolution of software is enabled by such product-line approaches is largely determined by the amount and ease of reuse of existing artefacts.

The maintenance phase of a product has always been significant and will increasingly be so [Lientz et al., 1978; Pigoski, 1996]. The growth of the complexity of systems is one reason [Lehman and Belady, 1985], the trend towards product families is another reason. From the survey in Chapter 3

[1]This chapter is based on: Van Dijk, Hylke W., Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, volume 3047 of *Lecture Notes on Computer Science*, pages 203–221. Springer-Verlag, 2005

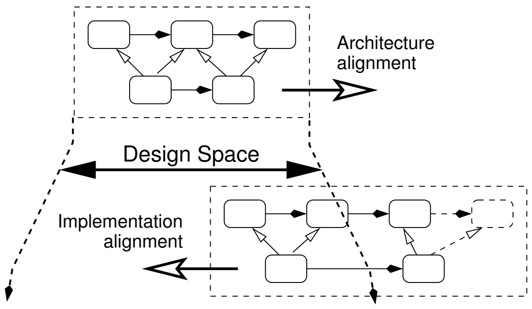


Figure 6.1: Aligning architecture and implementation

we learnt that new products are rarely developed from scratch and that reuse of existing development artefacts is typically ad hoc. These observations triggered our research in the field of conformance checking as a second step in enhancing the functionality of a product or adapting it to a changed environment (see Figure 1.2 on page 5). In our view a consistent set of software development artefacts is a prerequisite for successful reuse. Conformance checking is required to determine this consistency.

In general, conformance checking could be applied to all related artefacts produced in different phases of the software development process. In this chapter we focus on the conformance between software architectural views and the corresponding implementation.

An architectural view is a description of a software architecture that addresses a specific set of concerns. The guidelines for the creation of such views are described by associated viewpoints [IEEE-1471, 2000]. Views are developed during the architecture phase to specify constraints over design elements and relations [Perry and Wolf, 1992] for the subsequent product implementation, that is, the design space of Figure 6.1.

To check whether an implementation does not violate the constraints imposed by a view, views need to be created (reconstructed) from implementations [Van Deursen et al., 2004] to determine (predicate) the properties of the actual implementation from an architectural perspective, for instance, to determine how implementation units are related to each other. The constraints that an architecture specification defines effectively determine the bounds of the design space in which the implementation has to be realised. Figure 6.1 illustrates that the implementation in this case violates some of the architectural constraints and thus does not completely fall within the permitted design space. These violations can be resolved by either updating the architecture or the implementation.

Here, a technical and a conceptual problem arise. First the languages used in implementation (programming languages) and architectural views (modelling languages) differ. Second, the semantic gap between the elements and relations used in architectural views and the programming language constructs available to implement them makes it difficult to reconstruct an architectural view from an implementation. Therefore, for checking the conformance of an implementation with respect to an architecture, we propose to define a common conformance viewpoint at an intermediate level of abstraction. When architectural and implementation views are associated with a common viewpoint and use the same modelling language (i.e., metamodel) the identification of discrepancies between the intended and implemented architecture becomes possible.

We address the problem of conformance checking by means of a conformance checking system (CCS), describing the necessary steps. In order to be practically applicable in industry, it is required that such a framework builds on proven technology, and that its application is non-intrusive.

In this chapter we propose and experiment with a conformance check system (CCS) that facilitates conformance checks through the definition of a design-space conformance viewpoint bridging the semantic gap between the implementation and architecture. Such a viewpoint is to be derived from the involved architectural viewpoint in such a way that views associated with this viewpoint can both be extracted from the implementation and the architecture. We map the CCS to technology for model-driven engineering (MDE) [Bézivin, 2005], and apply it in an academic case study. Our experiments illustrate the definition of conformance viewpoints, comparing associated views, and visualisation of discrepancies between the intended (specified) architecture and the implemented (predicated) architecture.

This chapter is organised as follows. Our running example is introduced in Section 6.2. In Section 6.3 we present our CCS and explain what needs to be done to map it to MDE. Subsequently, we discuss the mapping of viewpoints to metamodels in Section 6.4 and the mappings and model transformations for conformance checking in Section 6.5. In Section 6.6 we discuss our CCS, the applied technology, and related work. We conclude in Section 6.8 with an overview of our contributions.

6.2 Running Example

The running example in this chapter is the development of an academic system: a digital music box (DMB) that reads data from a paper disc (the record). The disc contains a plotted spiral track of pulse-width modulated data bits. It rotates with a constant speed. The system tracks the spiral, reads the data bits, and then maps those bits to symbols. A string of symbols will be fed to an output device that transforms the string into audible music. Here, we focus on the process of reading the record and generating the symbol stream. A hardware view of this system is depicted in Figure 6.2. The system is composed of a traditional turn table and a set of simple light sensors that can be moved axially by a motor. The control is implemented in Java and distributed between a simple micro controller and a PC.

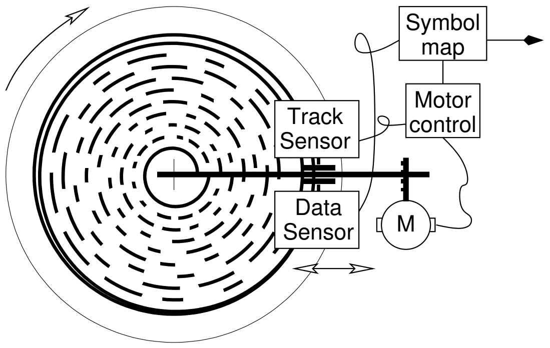


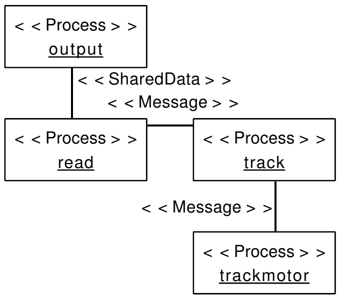
Figure 6.2: Digital music box reader system

The software architecture documentation consists of a set of views. Two of them are depicted in Figure 6.3 . Their governing viewpoints will be explained in more detail later.

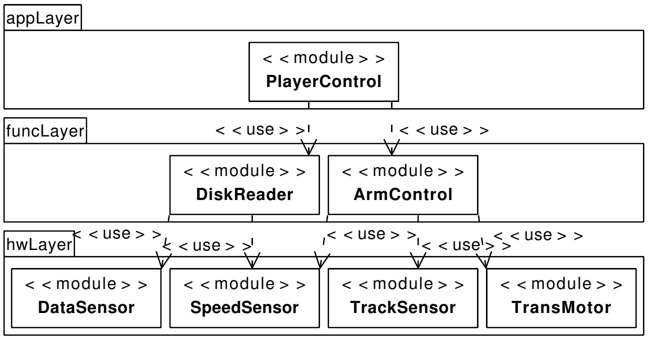
Figure 6.3(a) depicts the system as a set of communicating processes. This view uses stereotyped [OMG, 2007a] Unified Modeling Language[1] (UML) objects to represent components (processes) and stereotypes links to represent connectors (messages, and shared data). Here, the trackmotor process controls the speed of the motor that moves the sensors over the rotating disc (axially). The track process controls the Track Sensor and uses, via a Message connector, the trackmotor process to keep the Data Sensor positioned over the spiral track. The read process uses the Data Sensor to read the disc. Via a SharedData connector the output process plays the read bits on some output device. Finally, it was foreseen that the read and track processes needed to communicate via a Message connector, for instance to communicate the status of the read process.

The planned organisation of implementation units (modules) is given by the module-uses view depicted in Figure 6.3(b) . Here, modules are represented by stereotyped UML classes and uses relations by dependencies. Furthermore, a layering is represented using the package structure. Here, the lower layers are closer to the hardware (e.g., motors and sensors), while the upper layers are closer to the user.

[1]http://www.uml.org (June 2007)



(a) Communicating-processes



(b) Module-uses

Figure 6.3: Architectural views

6.3 Approach

6.3.1 Conformance Checking System

Our conformance checking system (CCS) is based on the idea not to compare architecture and implementation directly, but to derive views from implementation and architecture governed by a shared viewpoint expressed in the same language. This overcomes both the conceptual and technological problems mentioned in Section 6.1. As such, a conformance check between implementation and architecture that does not require changes to current ways of working (e.g., with respect to languages and views used) requires the definition of a *common viewpoint*. The semantics of such a "design-space conformance viewpoint" must be compatible with that of both architectural and implementation views. Thus, a *mapping* between the design-space conformance view the architecture and implementation must exist. Our approach is based on the work presented by Van Deursen et al. [2004] and Murphy et al. [1995] on architecture reconstruction.

In this chapter we consider the architectural view as leading. As in the approach by Murphy et al. [1995] there are three important situations for any element or relation in the implementation: *convergence*, *divergence*, and *absence*. A convergence indicates elements or relations in the implementation that have a corresponding element or relation in the architecture. A divergence only exists in the implementation and an absence only exists in the architecture. The result of a conformance check is a set of entities and relations that are attributed according to the three types above; the significance of mismatches (divergences and absences) found depends in general on the involved design decisions. Therefore, discovery of mismatches should serve as a trigger to investigate further if they are allowed and possibly documented elsewhere. If not, they are considered to be discrepancies that reduce the conceptual integrity [Brooks, Jr, 1975] of a system and may result in unexpected dependencies, reducing the system's maintainability.

The conformance checking system (CCS) outlined in Figure 6.4 is based on the process for architecture reconstruction presented by Van Deursen et al. [2004]. Using architecture and implementation artefacts, views are populated that are associated with a design-space conformance viewpoint. The conformance viewpoint is defined such that its distance to both the architectural and implementation viewpoints is minimised. Furthermore, it enables to attribute elements and relation with their conformance status in a subsequent comparison of the derived conformance views. For this a set of comparison rules is specified. Finally, a presentation filter visualises the comparison results.

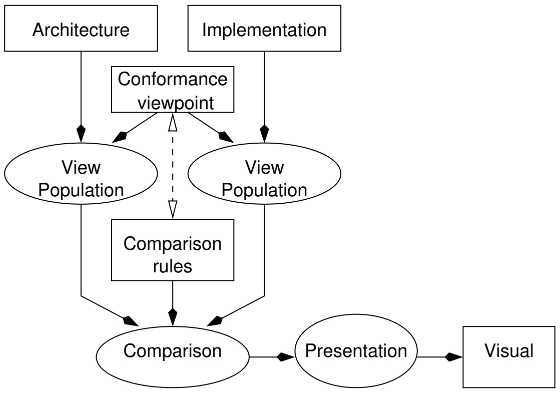


Figure 6.4: Conceptual Conformance Checking System

6.3.2 Model-Driven Conformance Checking

In the following sections we discuss how our CCS can be mapped to an MDE type of approach. Although, several types of technologies are available to do so, we will use technologies related to the Model Driven Architecture[1] (MDA). Alternatively, we could have used technologies based on the Extensible Markup Language[2] (XML) or on grammars. In fact, in earlier work we used XML technologies [Van Dijk et al., 2005]. There, we made the observation that specifications of models and transformations in XML tend to be verbose, and hence, difficult to maintain.

Using model transformation the XML syntax remains hidden. Moreover, we chose MDA because of the availability of languages and tools to manipulate architectural models specified using UML.

Key to MDE are models and (automatic) model transformations. As such, we confine our conformance check for an architectural view to its primary presentation. Here, we assume that this primary presentation consists of UML diagrams. Using model transformations we can then manipulate the underlying UML model in order to check the conformance of the corresponding implementation. To this end, we assume that viewpoints prescribe the modelling language to be used for the primary presentation of associated views.

In MDE, modelling languages are specified by metamodels. In the case of the architectural views that are the subject of our conformance check this typically is a subset of the UML metamodel. For the conformance viewpoints we derive these metamodels from the viewpoint descriptions.

[1]http://www.omg.org/mda (June 2007)
[2]http://www.w3.org/XML (June 2007)

In general, with MDE abstract models are transformed to more concrete models (and eventually into code). In our case we also transform models in the opposite direction to obtain a conformance models from the implementation.

In accordance with the MDA we use the MetaObject Facility[1] (MOF) for metamodelling and UML for modelling. For specifying the required model transformations we used the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL).

As such, we intend to check the conformance of an architectural model with respect to its corresponding implementation using MDA model transformations. Consequently, it is required to obtain an MDA type of model of the implementation.

Here, we use the concept of a technological space coined by Kurtev et al. [2002] and described as a working context with a set of associated concepts, body of knowledge, required skills, tools, and possibilities. MDA is one such technological space, based on MOF and model transformations. The MDA technological space can be referred to as modelware. Other technological spaces are the grammarware (based on grammars) and the XML technological space. Here, we refer to the translation of one technological space to another as projection or bridging. In particular, from the perspective of the target of such a projection we call it injection, and from the perspective of the source we call it extraction.

We, thus, obtain a model representation of the implementation by injecting it in a model based on an appropriate metamodel.

6.4 Viewpoints and Metamodels

The architectural views used to document a software architecture are associated with viewpoints. Several sets of architectural viewpoints have been defined. To attain a good coverage of the difficulties and possibilities of determining architectural conformance, we consider views from the two principal categories of views described in literature [Kruchten, 1995; Hofmeister et al., 1999; Clements et al., 2002a]: component-and-connector views and module views. As discussed in the previous section we derive a conformance metamodel for each of those viewpoints. In fact, these metamodels each specify an architecture description language (ADL).

Viewpoints define the restriction on the elements and relations to be used in their primary presentation. In our case we assume that diagrams are UML diagrams, and, thus, based on (a subset of) the UML metamodel. We derive its conformance metamodel from the restrictions it specifies. It

[1]http://www.omg.org/mof (June 2007)

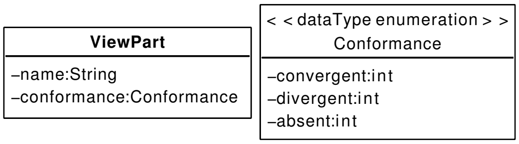


Figure 6.5: Generic metamodel element

is these elements and relations for which we want to establish conformance as convergent, divergent or absent.

First, we define a generic model element for conformance models in Figure 6.5. To avoid confusion, we refer to it as ViewPart. It includes a conformance attribute: an enumeration data type of convergent, divergent, absent. In the following we specialise ViewPart for concrete model elements for which we want to determine conformance in metamodels for particular viewpoints.

6.4.1 Component-and-Connector Views

Component-and-connector (C&C) views mainly address the question: how does the system work? The box-and-line diagrams created early during software design, usually are included in C&C type of views. C&C views are runtime views addressing concerns such as concurrency and flow of data. Most ADLs that have been defined are aimed at this type of views (see Medvidovic and Taylor [1997] for an overview). ADL models define the *structure* of a system in terms of runtime components and their interactions (connectors). Architectural components are loci of computation and state. Architectural connectors are loci of interaction [Shaw et al., 1995]. Both are architectural abstractions of elements that consume resources, either processing time or memory. As such, a complete C&C view is an abstraction a system's *structure* at runtime.

An example of a C&C view was depicted in Figure 6.3(a) on page 107. It shows concurrently executing components as communicating processes. The components interact through different types of connectors.

For the definition of a conformance metamodel for a C&C view we adhere to the terminology of ADLs (see, e.g., Garlan et al. [2000]). We refer to the ADL described in Figure 6.6(a) on page 113 as CPADL (Communicating-Processes ADL). The metamodel states that a System consists of sets of components and connectors. Each Component and Connector can interact with its environment through associated *interfaces*. In case of a component the interface is called a Port, whereas in case of connector we call it a Role.

In order to establish interaction between two components over a connector we can attach component ports to connector roles, with the limitation that such an Attachment is only allowed if the component interacts using the port as interface and according to the expectations described by the connector role, that is, port and role need to be compatible [Allen and Garlan, 1994]. All attachments together determine the configuration of the System. Note that although an Attachment is a relation, we define it as a 'first-class' model element (a ViewPart) to allow marking its conformance in a CPADL model.

For several elements we added specialisations for the specification of more specific communicating-processes views: OutputPort, InputPort, SinkRole, SourceRole, Process, SharedData, and Message.

Finally, the model elements for which we want to establish the conformance are defined as specialisations of the generic metamodel elements of Figure 6.5 on the preceding page.

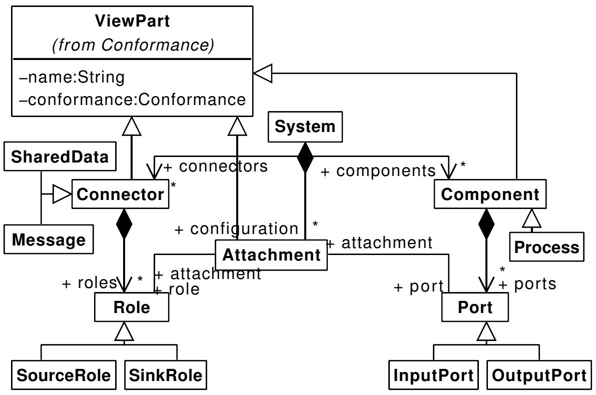
6.4.2 Module Views

In order to arrive at a system functioning as described by the C&C views, views are developed driving the actual *implementation* of software and hardware. Those views that capture the structural organisation of the implementation units are known as the set of *module views* and mainly address the question: how is the system developed? These views are used to divide the work among developers and development teams. Additionally, module views can be used to assess non-operational qualities of a system, such as modifiability. Figure 6.3(b) on page 107 displays a module view for the DMB system. It depicts the decomposition of the software implementation units (modules), their use dependencies, and layering.

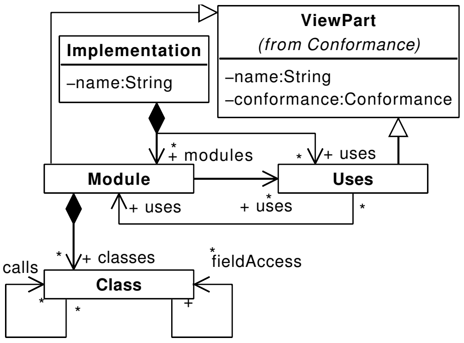
Modules are supposedly coherent units of functionality that are eventually assigned to development teams. Dependency relations between the modules of a development view are important. Several types of them exists, such as uses, allowed-to-use, and shares-data-with relations. Here, we focus on *use* dependencies.

The corresponding conformance metamodel is depicted in Figure 6.6(b) . Here, an Implementation consists of a set of modules and uses dependencies. A Module has a set of uses dependencies. In turn a Uses relation is associated with another Module. Finally, a Module consist of a set of classes. We added the concept of Class to simplify the reconstruction of a module-uses model from source code. Alternatively, we could have used separate metamodels for the result of the extraction step (classes and calls relations) on the one hand, and of the abstraction step (modules and uses relations) on the other.

Again, the model elements for which we want to check conformance are specialisations of relevant elements from Figure 6.5 on the previous page. We refer to the language defined by this metamodel as MADL (Module ADL).



(a) CPADL metamodel



(b) MADL metamodel

Figure 6.6: Metamodels

6.5 Mappings and Model Transformations

The model transformations involved in our CCS cover three phases: model population, conformance checking, and presentation. We discuss the transformations involved in each phase below.

6.5.1 Model Population

Model population involves injection of source artefacts into a model representation, which, subsequently is transformed in extraction and abstraction steps into a model that conforms to one of the defined conformance metamodels.

Injection

For the architectural UML models, injection is not required; they already are in the modelware technological space and, thus, can serve as source models in model transformations. To this end, UML tools can serialise the UML models to XML Metadata Interchange[1] (XMI) format, which, in turn, can be used by transformation tools.

In the case of the implementation, we first compiled a representation of the abstract syntax tree (AST) of the Java source code in JavaML [Badros, 2000], an XML based representation of Java source code. Similar technology is available for many other programming languages [Al-Ekram and Kontogiannis, 2005]. Then, we injected this XML document into a model representation conforming to a generic metamodel for XML. We could reuse both this injection and the XML metamodel, as they were already available from ATL's metamodel and transformation repositories[2].

Extraction and Abstraction

In extraction and abstraction steps the obtained implementation (XML-JavaML) model and the architectural (UML) model are each transformed into a model conform one of the conformance metamodels we defined.

Here, module and C&C views require a different approach. This is mainly due to the different relations between both types of architectural views on the one hand and the implementation on the other. The relation between module views and the implementation is a refinement relation, as these views are an abstraction of the implementation units that are to be (have been) delivered (cf. the relation between a class diagram and

[1]http://www.omg.org/mda/specs.htm#XMI (June 2007)

[2]http://www.eclipse.org/gmt/am3/zoos/atlanticZoo and http://www.eclipse.org/m2m/atl/atlTransformations (June 2007)

the corresponding object-oriented source code). The relation between C&C views and the implementation can be more complicated; these views are an abstraction of the systems structure at runtime and not of the implementation itself (cf. the relation between collaborating objects in a UML collaboration diagram and the source code that caused these objects to be instantiated). This makes reconstruction of those views a greater challenge.

Module Views For the population of a MADL model we have to identify modules and uses relations.

Typically, implementation-level modularisation constructs do not match one-to-one with the architecture-level modules. Developers typically have reasons to further refine the provided decomposition of the development views during implementation. In our approach we assume that the decomposition is recorded, for instance, through annotations.

A simple, yet sufficient, method is to add comments to the implementation with `@module(...)` clauses. These clauses associate a Java class (or other implementation unit) with a module of the architectural uses view. Alternatively, packages can be used to represent modules in an implementation. In this case, however, we already used packages to represent architectural layering.

Next to modules, a uses view defines uses relations [Clements et al., 2002a]. The notion of use has conflicting interpretations [Stevens, 2001]. In order to determine the existence or possibly inexistence of a particular uses relation, we start with the definition given by Clements et al. [2002a]: "Unit A is said to use unit B if A's correctness depends on a correct implementation of B being present." As our approach cannot guarantee that a module is correctly implemented, we take a pragmatic position by mapping the architectural uses relation to a checkable tuple: a link plus an action that effectuates the link. The link is a reference from a class that belongs to the 'using' module to the class that belongs to the 'used' module. The action can be anything from a function call to an attribute access. In fact, our interpretation captures calls and shares-data-with dependency relations, which are different specialisations of the depends-on relation.

We use the lifting operation described by Fahmy and Holt [2000b] to transform links, actions, and `@module` annotations to create uses relations in a target model.

First we recover classes and their dependencies from the JavaML representation of the AST. Using simple ATL expressions we select the XML Elements that represent a class and for each create a Class in the MADL target model. To instantiate corresponding calls and fieldAccess dependencies in the target model, we select all their child Elements representing method invocations and field accesses targeted at classes we also implemented.

Because JavaML discards comments, we use a simple *Perl*-script to retrieve `@module` clauses and generate an XML-document consisting of `<pair class=''...''module=''...''\>` elements. The projection of this document as XML model serves as additional source model to a transformation that creates modules that are composed of the classes created by the previous transformation. A final (refining) model transformation lifts the field access and call dependencies to the level of modules as Uses relations.

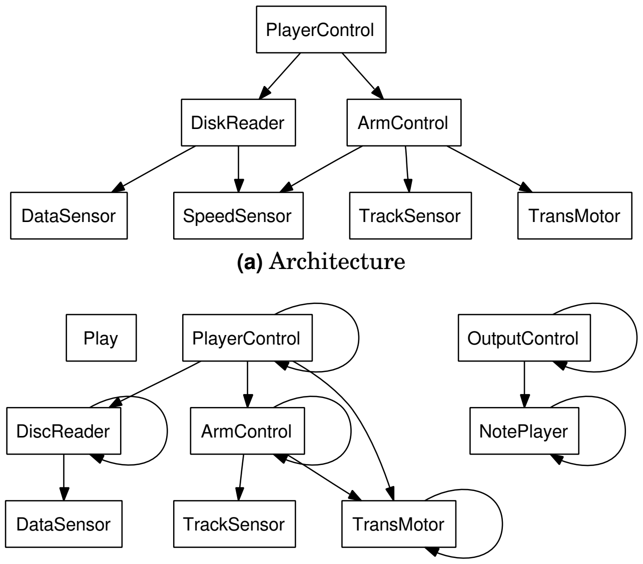
Next to the Java source code, the architectural uses view of Figure 6.3(b) on page 107 was another input for our CCS, representing the 'as-designed' architecture. Population of a MADL model from this UML model is straightforward because of the use of stereotypes to denote modules and uses relations.

The result (for implementation and architecture) is represented as a directed graph as shown in Figure 6.7 . Here, boxes represent modules, and arrows represent uses dependencies. It can be seen that the model derived from the implementation consists of two unconnected subgraphs. In fact, one subgraph corresponds to the part of the implementation deployed on the PC, the other to the part deployed on the micro controller. Obviously, no direct method invocations and field accesses are possible between those parts.

C&C Views Components, ports, connectors, and roles are architectural concepts that may or may not have explicit counterparts in the development views or implementation. The implementation is not merely a refinement of these architectural elements as in the case of development views. This makes the mapping between the architectural C&C views and implementation constructs indirect and more difficult.

The main concern of the C&C view in Figure 6.3(a) on page 107 is concurrency. For such a view the components correspond to implementation mechanisms for concurrency and parallelism, such as processes, threads and tasks. For example in the case of a system implemented in Java, these components correspond to threads. Similarly, connectors, in that case, are abstractions of the mechanisms that allow these threads to interact, for instance inter-process-communication mechanisms, remote-procedure calls, or shared-data.

Creating a C&C view from static sources is very application specific; it depends on conventions for implementing, for instance, concurrency and communication mechanisms. In this case we want to reconstruct a CPADL model representing a set of communicating processes. As said, in Java this type of components corresponds to threads. Java threads are classes with a `main`-method or classes that extend the Java `Thread` class. In this



(a) Architecture

(b) Implementation

Figure 6.7: Reconstructed MADL models

case we also assume that such classes are instantiated only once. Therefore, the first transformation step identifies these classes in the JavaML model. For each we create a Process component in the CPADL target model. Note that because in this case there exists a one-to-one mapping between an implementation construct (Java thread) and the components in a communicating-processes view, it is not necessary to rely on annotations as for the module view.

In addition, we search in the first transformation step for two types of interactions: message passing and shared-data. These types of connectors were implemented using method invocations and Java streams, respectively.

For identification of the relevant method invocations we largely reuse the ATL expressions for the population of the MADL model. Only now we just consider method invocations between identified threads. For each distinct method invocation between two threads we create a Message connector, and, on the side of the source of the message, an OutputPort for the involved component and a SinkRole for the connector; at the 'target side' we create an InputPort and a SourceRole. Finally, we create Attachments for those ports and roles.

The Java platform for the micro controller offers a special type of (buffered) stream class for which we create a SharedData connector in the CPADL target model. By identifying each thread that reads from or writes to an instance of that class we identify Ports (input or output), Roles (source or sink), and Attachments.

Identification of Message connectors in the transformation explained above, results in a separate connector instance for each distinct method invocation between two threads. In general, connectors describe a complete protocol for the interactions allowed between two components. Therefore, using another model transformation, we abstract all Message connectors between two components into a single connector representing the allowed message-based interaction between those components.

The resulting graph is given in Figure 6.8(b) . We represent components by rectangles and connectors by ellipses. Attachments are represented by edges and their arrowheads indicate the 'direction' of the involved ports and roles.

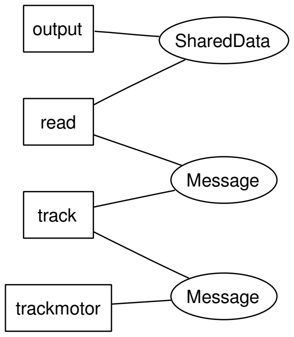
The architectural input was the C&C view of Figure 6.3(a) on page 107. To turn this view into a CPADL model, we map each stereotyped Object to a Process in the CPADL target model. Each Link is mapped to an appropriate Connector depending on the applied stereotype. Ports, Roles, and Attachments are created where necessary. Note that we cannot distinguish between in- or output ports or source or sink connectors here, as no direction is provided. The CPADL model populated from the architecture is given in Figure 6.8(a) .

6.5.2 Model Comparison

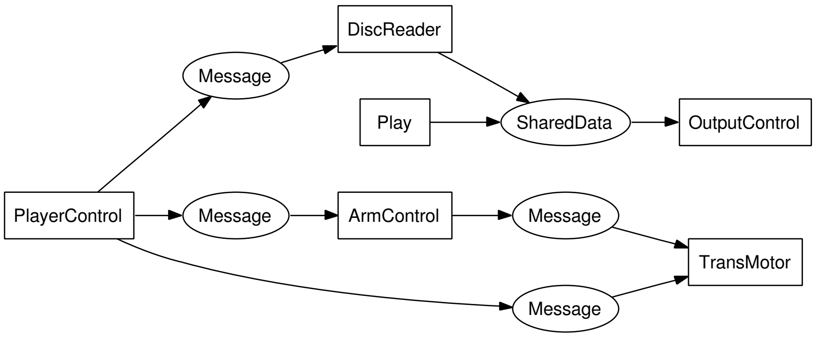
The comparison transformations have two source models (derived from implementation and from architecture) and generate an attributed target model, containing the elements of the source models attributed with one of the following labels: convergent, divergent, or absent.

Module Views For each module in both source models it is checked whether it is a convergent, divergent or absent module. This is simply done using name matching.

A uses relation in the implementation source model is considered convergent if a uses relation exists between two modules in the architecture source model that correspond with the two modules involved in that uses relation in the implementation source model. The result is visualised in Figure 6.9 . Convergent, divergent, and absent entities are represented by solid, dashed, and dotted lines respectively.



(a) Architecture



(b) Implementation

Figure 6.8: Reconstructed CPADL models

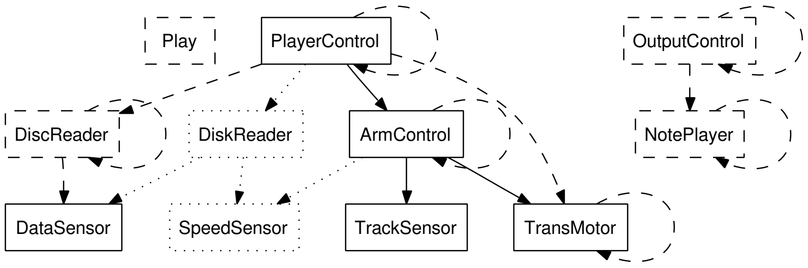


Figure 6.9: Merged MADL conformance model

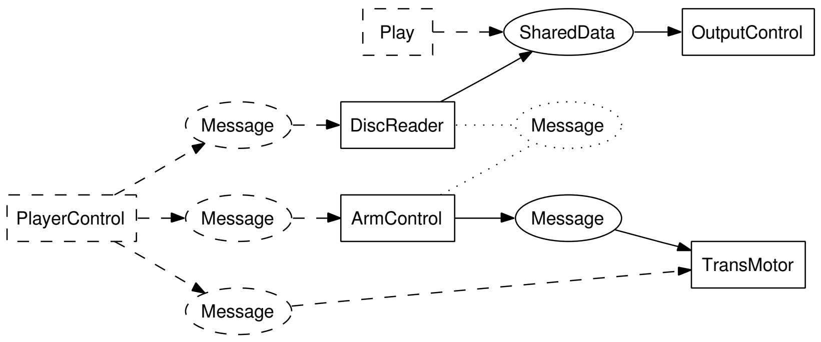


Figure 6.10: Merged CPADL conformance model

Note that we determine manually which mismatches actually involve discrepancies. Partly, the identified mismatches originate from naming, e.g. DiskReader and DiscReader. One entity has not been implemented: the SpeedSensor. A divergent uses relation emerges between PlayerControl and TransMotor. In fact, this relation violates the intended layering.

C&C view Comparing the generated CPADL models results in the identification of convergences, divergences, and absences as shown in Figure 6.10. Comparing the C&C runtime views of Figure 6.8 on the preceding page involves merging the namespaces. Especially in the case of a C&C view this is necessary, as the names in the source code from which we reconstruct the C&C view are derived from the module views and therefore often do not match those in the C&C view. To remedy this we allow the user of our transformation to provide a mapping between the two involved name spaces. Again, we use XML to make this possible. In this case the mapping consists of

```
<map name="CC">
  <mapping src="DiscReader" arch="read"/>
  <mapping src="OutputControl" arch="output"/>
  <mapping src="ArmControl" arch="track"/>
  <mapping src="TransMotor" arch="trackmotor"/>
</map>
```

As can be seen in Figure 6.10, the PlayerControl component is a divergence. Consultation with the architect revealed that it was intended as a connector between the read and track components. However, in the implementation it included handling user interaction for which it required a separate thread.

6.5.3 Model Presentation

The final step in our CCS is the presentation of the generated conformance models. As the defined MADL and CPADL metamodel only define the structure (abstract syntax) of the associated models and not their graphical notation (concrete syntax) additional transformations are necessary to a language that has an associated graphical notation.

We used the graph description language DOT[1] to visualise the result. Although DOT is a textual (grammar-based) language, a MOF-based metamodel is also available, allowing processing using ATL. We defined transformations from CPADL and MADL to the DOT metamodel. Finally, the textual representation of the generated DOT models was extracted by a special type of transformation. This transformation queries a DOT source model and generates corresponding DOT code as output. The result can be visualised using *dot*. Examples were depicted in Figures 6.7 to 6.10 on pages 117–120.

6.6 Discussion

Below we address a number of issues related to the use of MDE for our CCS. Subsequently, we discuss a number of potential improvements of the approach.

6.6.1 Modelware

Model Population In our experiment we implemented the population of conformance models via XML. We used existing bridges from grammarware to XML (JavaML) and from XML to the modelware technological space (the injector for XML data). The drawback of using these bridges, is that subsequent transformations that populate the MADL and CPADL models are specified in terms of XML metamodel elements (e.g., Node, Element, Attribute), rather than elements specific for Java (e.g., Class, Method, Field). This makes specifying these transformations prone to errors. A helper operation to manipulate a class, for instance, is specified in the context of XML elements that represent classes. Instead of using the type system, it has to be checked explicitly that an XML element indeed represents a class.

By construction of an ATL library of helper operations related to the XML model representation of Java (e.g., an operation to collect all XML elements that represent a method invocation for an XML element that represents a class) we could raise the abstraction level somewhat. However, ideally, we would like to use a representation based on a 'real' Java metamodel. In fact, such a metamodel is currently available from the ATL metamodel

[1]http://www.graphviz.org (June 2007)

repository. The problem remains to populate a model conform this metamodel based on a set of Java source files. A solution to this problem could be a transformation that transforms XML models (based on JavaML) into models based on the Java metamodel. This would allow to specify the model extraction at the desired level of abstraction. In general, such a transformation would also be useful for other software evolution tasks that involve Java source code and might be solved in the modelware technological space (e.g., architecture reconstruction, program understanding, metric calculations).

Modelware Management With the application of model transformations to more and more complicated and diverse problems (see, e.g., Chapters 5, 7, and 8 of this thesis), managing the involved model artefacts becomes an issue.

Although we did not discuss all of them, the approach presented in this chapter already involved seven different metamodels, 11 different transformations, and over 20 source, target, and intermediate (generated) models. In addition, the complete solution involved several transformations executed outside the modelware space using tools such as *sed*, *Perl*, and *java2xml* (for transformation of a Java program text into a JavaML representation). Currently, we manage all these artefacts using the Java build tool Ant[1], for which the ATL development tools provide special tasks to execute transformations, save and load (meta)models, and apply projectors. Using Ant, we completely automated our approach, including compilation of an XML representation for Java source code, execution of various model transformations, and generation of PostScript output for DOT graphs.

Modelware Reuse Fortunately, not all of the modelware artefacts mentioned above have to be developed from scratch. We reused and adapted metamodels from ATL's metamodel repository (e.g., metamodels for DOT and XML). Also projectors were reused (e.g., the XML injector, and the DOT to text extractor).

Still specification of all remaining metamodels and model transformations takes considerable effort. However, transformations defined for the model transformation phase for injection of Java source code into a model representation can be reused for conformance checking of other views, as well as for different types of applications that involve the manipulation of programs.

The metamodels and transformations used in the comparison and presentation phases are specific to a particular viewpoint. On the other hand these metamodels are easily extended for other viewpoints, for instance, by

[1]http://ant.apache.org/ (June 2007)

addition of additional types of connectors or dependency relations. If such additions do not require a specific approach for comparison and presentation, the transformations we specified can still be applied.

6.6.2 Improving the Approach

Generalisation The use of ATL makes our approach specific to the MDA technological space. This means that we require source models to conform to a MOF-based metamodel and to be serialised with XMI. In principle this makes our approach compatible with most UML tools (via XMI). However, in practice additional transformations are often required to exchange source and target models between modelling, visualisation and transformation tools (see also Sections 5.7 and 7.9).

The specification of model population transformations is dependent on the applied (programming and modelling) style (e.g., usage of patterns and coding conventions). The required complexity of transformation rules also depends on the programming style. For instance, the use of getter and setter methods circumvents the need to look for direct field access.

To generalise the comparison step of our approach we considered the definition of a complete generic metamodel for (conformance) views. Such a metamodel would define additional ViewParts (see Figure 6.5 on page 111) such as Element, ConnectingElement, and Relation. These ViewParts can then be specialised by metamodels for particular viewpoints. For example, in the CPADL metamodel a Connector would be a ConnectingElement, and an Attachment a Relation. All this would make sense when it is possible to specify transformations for comparison and presentation in terms of those generic ViewParts. These transformations can then be used for arbitrary conformance models. The problem with this approach is that it is not possible to also define generic relations between metamodel elements that can be specialised by a concrete metamodel. So although it is possible to develop a transformation to always present a ConnectingElement as an ellipse, such a transformation cannot also instantiate its relations in a generic way.

We can overcome this problem by the use of higher-order transformations. These are transformations that have another transformation as source or target model. For ATL this is made possible by the availability of a metamodel for model-based representation of ATL transformations and corresponding projectors. Such a higher-order transformation would take the metamodel involved in the conformance check as source model and produce a model of the transformations for comparison and presentation of the associated models. This appears to be feasible considering the fact that the transformations we implemented for those steps are quite similar.

This approach is investigated in Graaf and van Deursen [2007b], where we provide a more extensive generic metamodel, as discussed above. Based on this (abstract) metamodel we define concrete metamodels for particular views. The higher-order transformation generates helpers and transformation rules for concrete instances of the abstract metamodel elements in a particular view's metamodel. The resulting ATL transformation is applied to models conforming to the concrete metamodel to check their conformance.

Here, a trade-off can be identified. Addition of extra details in a generic metamodel (e.g., relations between metamodel elements), allows more transformation code to be reused, for instance, for visualisation of conformance models with DOT. On the other hand, this prevents to make the conformance check and visualisations specific for a particular metamodel. For instance, we visualised connecting elements using ellipses and other elements using boxes. In a different situation it might be necessary to visualise different types of connecting elements (message or shared data) differently.

Identifying Discrepancies In the module view example several mismatches occurred due to naming (e.g., DiskReader vs. DiscReader). This can be solved by allowing a user to supply a mapping between name spaces in a similar way as was done in the case of the C&C view. For determining whether other mismatches are discrepancies, design decisions have to be considered.

In general, defining what is a mismatch depends on the type of views involved and the intentions of the architects. We did, for instance, not report mismatches for attachments related to different port types when a port in the model derived from the implementation is a specialisation of a corresponding port in the model derived from the architectural view.

Static conformance checking of runtime views Although C&C views describe a system at runtime, we only used static information for our conformance check. As a result, we cannot always be sure that, for instance, two identified attachments connect the same component and connector instances. It would be interesting to also consider dynamic information. This raises questions such as how to inject that information into models, and what type of metamodels are suitable for that. One possibility would be to use traces to instantiate message sequence chart type of models, as is done by Cornelissen et al. [2007].

Introducing annotations Although we required our approach to be non-intrusive, we did introduce the use of annotations to register more detailed decompositions than prescribed by the module view. Assuming that the module views indeed drive the implementation activities, the advent of integrated development environments makes dealing with such annotations straightforward. Eclipse[1] could, for instance, be easily changed such that this information is requested from the programmer in the wizard for defining a new class. Furthermore, although we use Java 1.4, in version 1.5 annotations have become an integral part of the Java language. Subsequently, this information could be included in the header of the template used by the wizard to create classes.

6.7 Related work

Krikhaar [1999] and Mens [2000] independently compared a number of approaches to check architecture conformance. However, conformance between models at different abstraction levels is not addressed. Moreover, most approaches dictate the introduction of specific modelling languages, requiring a change to current ways of working.

They both mention Murphy et al. [1995] that introduces software reflexion models. In that work a high-level model is combined with a source model and a user provided mapping between the two to generate a so-called reflexion model. This model indicates where source model and high-level model agree. Although our merged conformance model is clearly based on their reflexion model, they only indicate conformance for relations. Their approach is more suited for cases when the semantic gap between architecture and implementation is very large.

Our approach extends the generic process for architecture reconstruction proposed by Van Deursen et al. [2004]. Their process is based on several industrial case studies and includes separate steps for data gathering, knowledge inference, and information interpretation. We extended this process for conformance checking by addition of a comparison step. Furthermore, we make architectural viewpoints concrete by the definition of metamodels.

Han et al. [2003] discuss the steps required for the reconstruction of web applications. Although their approach is not automated, their uses relation is closer to the definition of Clements et al. [2002b] than the one we implemented. Next to uses relationships based on method calling they also consider such relationships based on another type of logical interface, HTTP request parameters. Furthermore, they introduce the 'knows' relation, a weaker type of uses. The latter we could easily introduce by generating 'knows' relationships between elements that own a link (i.e., reference) to each other that is not effectuated (i.e., by a method call).

[1]Eclipse is a widely-used, open-source integrated development environment, see http://www.eclipse.org (June 2007)

An alternative to checking conformance after development, would involve the use of MDE to generate source code or extend the implementation language with architectural constructs as was done in ArchJava [Aldrich et al., 2002]. Such approaches directly connect architecture to implementation, improving consistency. However, this requires at least a change in the way of working of the implementation phase, for instance the use of a new language. This poses a barrier for implementing such an approach in practical settings.

6.8 Conclusions

In this chapter we propose a conformance checking system (CCS) to systematically determine discrepancies between an intended architecture and the realised architecture. Illuminating these differences is a preparatory step for architecture migration in which previously developed artefacts are reused for reasons of efficiency. Our the conformance checking system (CCS) is non-intrusive. It coordinates the interaction between the architecture and the implementation domain of expertise, while regarding them autonomously. It uses readily available, possibly tailored, technology for the actual implementation of CCS. As such, the main contributions of this chapter are:

- a generic process for conformance checking of architectural views;
- extensible metamodels for C&C and module viewpoints, including mappings from implementation and architectural artefacts; and
- a demonstration of how to combine different types of technologies (inside and outside the modelware technological space) into an integrated approach, while reusing several existing metamodels and transformations

Although our approach is largely automated, checking the conformance for particular type of views might require a specific approach. Still, the CCS provides a generic process based on a common design-space viewpoint. The CCS relies on a clear definition of associated metamodel and the mappings from the architectural and implementation artefacts to this common metamodel.

The design-space metamodel (e.g., Figure 6.6 on page 113) captures *checkable* concepts, which are the consensus between verifying abstract properties of the architecture and emerging properties of the implementation. Possible discrepancies between the two are revealed as mismatches between the derived conformance models and the impact of a mismatch.

We gave examples of conformance metamodels for the two principal categories of views and the mappings from architecture and implementation artefacts. In our case study we used and configured MDE technology.

Although the results are promising we encountered intriguing research questions, such as to what extent we can further generalise the approach (i.e., the involved metamodels and transformations). Here, higher-order transformations and the use of reflection are two possibilities we will investigate in the future.

To get a better understanding of the scalability of the approach we intend to apply the proposed approach on an industrial case. An interesting possibility is the application of this approach for checking the conformance of the behavioural specifications discussed in Chapter 5 of this thesis. For this we have to investigate whether our approach can be applied to automate the manual comparison step, in which the state machine we generate from a set of scenarios are compared to the state machine used for code generation.

Chapter 7

Model-driven Migration of Supervisory Machine Control Architectures[1]

Supervisory machine control is the high-level control in advanced manufacturing machines that is responsible for the coordination of manufacturing activities. Traditionally, the design of such control systems is based on finite state machines. An alternative, more flexible approach is based on task-resource models. This chapter describes an approach for the migration of supervisory machine control architectures towards this alternative approach. We propose a generic migration approach based on model transformations that includes normalisation of legacy architectures before their actual transformation. To this end, we identify a number of key concerns for supervisory machine control and a corresponding normalised design idiom. As such, our migration approach constitutes a series of model transformations, for which we define transformation rules. We illustrate the applicability of this model-driven approach by migrating (part of) the supervisory control architecture of an advanced manufacturing machine: a wafer scanner developed by ASML. This migration, towards a product-line architecture, includes a change in architectural paradigm from finite state machines to task-resource systems.

7.1 Introduction

As software intensive systems evolve they tend to become increasingly complex [Lehman and Belady, 1985]. Furthermore, the architecture documentation and its corresponding implementation tend to follow asynchronous

[1]This chapter was published earlier as: Graaf, Bas, Sven Weber, and Arie van Deursen. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2007. Doi: 10.1016/j.jss.2007.06.007

evolutionary paths. Consequently, the conformance between the architecture specification and software implementation decreases as a software system evolves [Bril et al., 2005].

In practice, increased complexity and loss of conformance between the architecture as intended and the architecture as implemented make a system more difficult to change [Perry and Wolf, 1992]. This results in an increase of both development and maintenance effort. The involved effort can, for instance, be reduced by the separation of concerns, the use of product-line architectures, model-driven development and automatic code generation.

In this chapter we consider the migration of supervisory machine control (SMC) architectures towards a product-line approach that, amongst others, supports model-driven development and code generation. In practice, adopting such techniques requires architectural changes. When migrating towards a product line, such a migration needs to be applied repeatedly to migrate different product versions into product-line members. Therefore, ideally, one would like to make such a migration reproducible by automatically transforming one architecture into another. In this chapter we investigate how this can be done using model transformations. Developing a model-driven migration approach is particularly beneficial in a setting where product migration is not a one-off exercise.

In an advanced manufacturing machine, supervisory control [Ramadge and Wonham, 1987; Gohari and Wonham, 2003] is responsible for the coordination of the (discrete) high-level machine behaviour. This requires, amongst others, interpretation of manufacturing requests, synchronisation, scheduling, conditional execution, and exploitation of concurrency with respect to the resulting manufacturing activities [Sabuncuoglu and Bayiz, 2000; Buttazzo, 2002; Reveliotis, 2005]. For advanced manufacturing machines, the control systems have an indicative order of magnitude of 10 SMC components, each encompassing 10^4 - 10^5 lines of code.

This chapter was motivated by the prototype migration of the SMC architecture of a wafer scanner as developed by ASML, a manufacturer of equipment for the semiconductor industry. We use this wafer scanner as a running example to illustrate the migration of a legacy architecture, based on finite state machines (FSMs), to a new architecture that is based on task-resource systems (TRSs). This migration is spurred by the fact that a TRS-based SMC architecture, as opposed to an FSM-based one, is declarative, separates concerns, and supports run-time dependent decisions [Van den Nieuwelaar, 2004]. As a result, the maintainability and flexibility of the migrated software systems is improved.

We consider the start and end point of the migration as different architectural views [IEEE-1471, 2000]. We refer to these views as the source and target view respectively. An important element of an architectural view is its primary presentation [Clements et al., 2002a], which typically contains one or more diagram(s). In this chapter we focus on the models and their governing metamodels underlying those diagrams. In our migration approach we use these models to consolidate and reuse as much existing design knowledge as possible. As such, we consider migration to constitute a series of model transformations, which we implemented using Model Driven Architecture[1] (MDA). It should be noted that we only consider the actual migration approach; the paradigms for the migration start point and end point are prescribed by our industrial case.

In order to define a reproducible mapping and perform the migration, we define practical transformation rules in terms of patterns associated with the source and target metamodels. These transformation rules are practical in the sense that they are based on an actual migration as performed manually by an expert. Based on this migration, we have formulated generic, concern-based transformation rules. These rules are defined using a model transformation language making our approach automated. Due to practical reasons, which are mainly associated with the informal use of modelling languages in industry (see Chapter 3 and Lange et al. [2006]), we first normalise the legacy models before applying our model transformations.

Although we focus on the migration of the SMC architecture of a particular manufacturing system, the ASML wafer scanner, the contributions of this chapter are applicable to similar (paradigm) migrations of supervisory control components in general. The presented industrial results serve as a proof a concept, additional migrations have to be performed before the results can be properly quantified. The experiences as outlined in this chapter are, to a lesser extent, relevant for all software architecture migrations that can be seen as model transformation problems.

The remainder of this chapter is structured as follows. Section 7.2 discusses related work. In Section 7.3 we introduce SMC, concerns specific to SMC systems, and our running example. A generic migration approach, which we use for the migration between the introduced architectural paradigms, is presented in Section 7.4. The source paradigm of the migration and the normalisation of its associated views are discussed in Sections 7.5 and 7.6. The target paradigm and our transformation rules are treated in Sections 7.7 and 7.8. We illustrate each step of the migration by means of a running example. Section 7.9 reflects on the migration results. Finally, we conclude in Section 7.10 with a summary of contributions and an overview of future work.

[1]http://www.omg.org/mda (June 2007)

7.2 Related Work

The process that we propose considers migration as a mapping from a source to a target view. This approach is inspired by the approach for architecture reconstruction as described by Van Deursen et al. [2004]. There, architecture reconstruction is considered to be a mapping from a source view that is extracted from code to an architectural target view.

Our process can also be seen as the application of the MDA to software *migration* rather than to software *development*. In the MDA, software development is conceived as a series of transformations from source models to target models. As such, in both processes, model transformations are applied but in our case an essential normalisation step is added to the original MDA framework.

Fahmy and Holt [2000a,b] discuss several types of generic architecture transformations that can be viewed as graph transformations. In this chapter we consider domain-specific transformations on architectural models that are more complex than typed graphs; next to typed nodes, our models also include attributes on nodes and edges. Moreover, their transformations are intended for small, evolutionary changes to a software architecture, whereas the transformations as discussed in this chapter are driven by the migration to a different architectural paradigm.

Bosch and Molin [1999] use architecture transformations during architecture design to realise the non-functional quality requirements of a system. Of the transformation types they identify, the application of an architectural style is closest to our work. To some extent, changing the architectural paradigm from FSMs to TRSs, as considered in this chapter, could be understood as such a transformation. In our case, however, this transformation also results in a product-line architecture.

In other work, transformations are applied to the migration of software at the level of source code. Baxter et al. [2004] present a toolkit that uses generalised compiler technology for this purpose. Gray et al. [2004] use this toolkit for model-driven program transformations where vertical and horizontal transformations are identified. Here, vertical transformations concern the creation of software artefacts from artefacts at different abstraction levels (translation). Application of the MDA typically involves vertical transformations, whereas they investigate its applicability to horizontal transformations. The architecture migration we discuss can also be considered a horizontal transformation. However, where they focus on the source code, we consider migration at the design level.

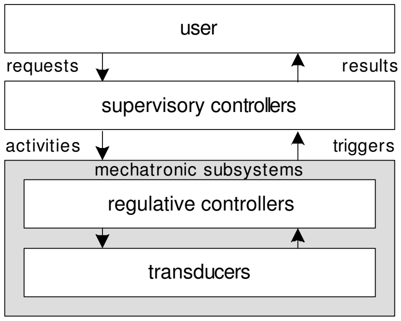


Figure 7.1: Machine control context

7.3 Migration Context

In this section we first define the SMC context. Next, we introduce the motivating case and running example for this chapter: a typical wafer scanner as produced, for instance, by ASML. In this setting we briefly discuss the key concerns for SMC systems in general. These concerns need to be addressed during architecture migration. As such, they form the basis for the design of our normalisation and transformation rules.

7.3.1 Supervisory Machine Control

The machine control context is clarified in Figure 7.1. From a supervisory perspective, (sub)frames, transducers and associated regulative controllers form mechatronic subsystems that execute manufacturing activities to add value to products. The recipe- and customer-dependent routing of multi-product flows, with varying optimisation criteria, constitutes one of the key (supervisory) control issues. Moreover, advanced manufacturing machines must respond correctly and reproducibly to manufacturing requests, run-time events and results. Consequently, to interpret manufacturing requests and to ensure feasible machine behaviour, a supervisory machine control component is required to coordinate the execution of manufacturing activities [Ramadge and Wonham, 1987; Sabuncuoglu and Bayiz, 2000; Van den Nieuwelaar, 2004].

In practice, a high-level manufacturing request is translated into valid low-level machine behaviour using multiple, consecutive control-layers. This is supported by recursive application of the control context from Figure 7.1: manufacturing activities of one level become manufacturing requests for the next level until the level of the mechatronic subsystems.

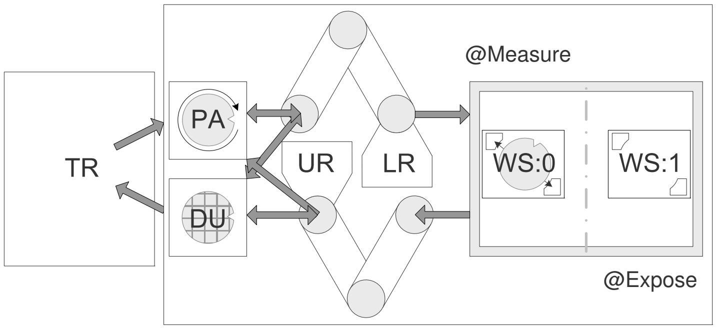


Figure 7.2: Simplified layout of a wafer scanner

7.3.2 Running Example: A Wafer Scanner

In this chapter we consider the ASML wafer scanner as a representative example of an advanced manufacturing machine. Wafer scanners are used in the semiconductor industry and perform the most critical step in the manufacturing process of integrated circuits (ICs). Figure 7.2 illustrates a scanner and its subsystems.

A neighbouring machine, the track (TR), performs pre-processing steps and delivers silicon wafers to the pre-alignment system (PA), where the wafer orientation and alignment are determined and adjusted. Next, the load robot (LR) transports the wafer to one of the two wafer stages (WS:0 or WS:1). Here, the wafer characteristics are measured. After measurement, the wafer stages are swapped and the measured wafer is exposed. During exposure, a laser projects an image of the required IC pattern onto the wafer's surface through a demagnification lens. A wafer is exposed in a scanning fashion, similar to the process used in a photo-copier. Eventually, the wafer comes to hold hundreds of small copies (i.e., dies) of this pattern.

After exposure, the stages swap back and the unload robot (UR) transports the exposed wafer to the discharge unit (DU) where it is buffered. Next, the wafer is picked up by the track again to undergo various post-processing steps. Now, the wafer is ready for another exposure if needed; the process is re-entrant. With each passing, another layer is added to each die. Once the wafer has been fully processed and inspected, it is diced into individual dies that are packaged to form ICs such as microprocessors.

For the SMC component of the wafer scanner depicted in Figure 7.2, we can identify the 'process wafer *w*' manufacturing request, which supports concurrent measuring and exposing of two wafers. To perform this request manufacturing activities such as 'load wafer *w* onto wafer stage WS:0' and 'unload wafer *w* from wafer stage WS:1' are executed. For instance, after a wafer has been exposed, and the stages have swapped, the wafer must be unloaded from its stage. In turn, these activities are requests for a lower-level SMC component. In this chapter we will use the 'process wafer' and 'unload wafer' requests as illustrative examples.

7.3.3 Concerns for Supervisory Machine Control Systems

In advanced manufacturing machines, multiple manufacturing activities - and sequences hereof - may fulfil a particular request and, in turn, multiple mechatronic subsystems may be available to perform a particular activity. That is, multiple alternatives exist that require the selection of a specific subset of both manufacturing activities and mechatronic subsystems to fulfil a given manufacturing request. For instance, when considering Figure 7.2 , removing a wafer from DU can be done using either UR or LR. For supervisory control of advanced manufacturing machines in general, the following key concerns are identified.

The execution of an activity on a selected subsystem implies a specific physical state transition of that subsystem. The selected sequence of activities for a subsystem requires matching end states and begin states of consecutive state transitions. When these states do not match, an additional transition, a setup, has to be executed between consecutive activities. For instance, when UR is idle at PA, a rotation has to be performed before a wafer can be unloaded. In SMC, these *sequence-dependent setups* are common.

Intuitively, controlled *usage* of mechatronic subsystems is another important concern. The control system generally checks the availability of a subsystem that is required for a manufacturing activity. Once available, the subsystem should be effectively claimed for the given activity. When an activity has been (co)performed by claimed mechatronic subsystem(s), all should be unclaimed or released. In our wafer scanner example, the unloading of a wafer requires both UR and, for instance, WS:0.

In order to take full advantage of installed capacity, *concurrent execution* of activities is done where possible. In practice, activities can be executed concurrently unless this is explicitly prohibited by precedence (sequence) relations between manufacturing activities or usage of the required mechatronic subsystems. In our wafer scanner example, one wafer can be measured and prepared for exposure while another wafer is being exposed.

Synchronous execution is another common concern. This not only refers to synchronisation of activities such that they are executed one after the other (e.g., load a wafer before processing it). It also applies to synchronisation of specific subsystem state transitions related to two activities. For instance, physical space is often limited, resulting in multiple mechatronic subsystems that simultaneously operate within a confined space.

This results in so-called hazardous areas in which subsystems can collide and state transitions must be induced synchronously to ensure safety (e.g., swapping WS:0 and WS:1).

Finally, *conditional execution* of manufacturing activities needs to be supported. That is, depending on certain conditions in a machine, different execution paths for a manufacturing request might be activated, each consisting of consecutive manufacturing activities. An example of such a condition in our wafer scanner example is the presence of another wafer on the wafer stage at the measure-side.

During migration, sequence-dependent setups, subsystem usage, concurrent execution, synchronous execution and conditional execution are concerns that need to be addressed. To this end, we defined concern-based transformation rules that map these concerns from the legacy to the new architecture.

7.4 Model-Driven Migration

Ideally, the migration of software architectures is complete, reproducible, reliable and automated. We consider the start and end point of the migration as different architectural views, referred to as the source and target view respectively. This is similar to the approach for architecture reconstruction as described by Van Deursen et al. [2004]. An architecture view is associated with a viewpoint [IEEE-1471, 2000], that, amongst others, specifies a metamodel for models underlying the primary presentation [Clements et al., 2002a] of that view. In this chapter we focus on those models.

For the migration of source models into target models we propose the migration approach as shown in Figure 7.3 . It uses a two-step process that includes a normalisation and transformation step.

Models and their specifications are often incomplete and have a tendency to become inconsistent and ambiguous over time. This makes directly translating a source model into a target model inherently difficult. This is amplified further by tool limitations and the generally informal use of modelling paradigms and languages in industry (see Chapter 3 and Lange et al. [2006]). Combined with incomplete or generic metamodels (e.g., the Unified Modeling Language[1] (UML) metamodel), or no explicit metamodels at all, a multitude of models becomes conceivable that all have the same intended meaning.

In fact, an analysis of how SMC concerns are addressed in the source models for our migration, revealed a large variation in the used idiom. This

[1]http://www.uml.org (June 2007)

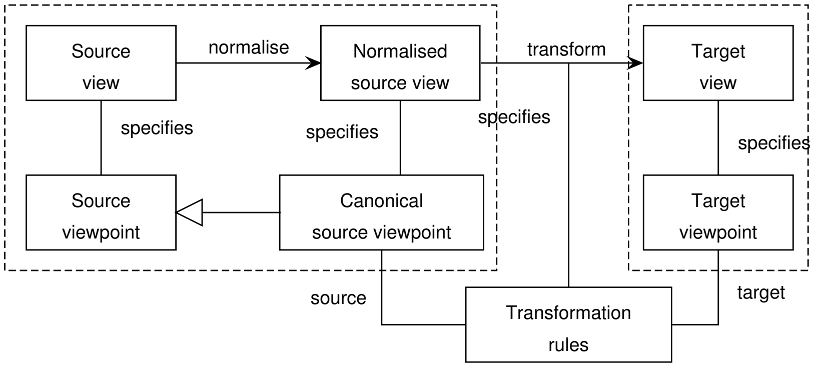


Figure 7.3: Generic two-phased migration approach

makes it infeasible to specify generic corresponding transformation rules. As such, we introduce an intermediate normalisation step that uses a set of normalisation rules to obtain a normalised source model. The normalisation rules are defined as mappings from the source metamodel to the normalised source metamodel. This normalised metamodel describes a subset of the models described by the source metamodel. Next, a set of transformation rules can be applied to transform a normalised source model into the target model. These transformation rules are defined as mappings from the normalised source metamodel to the target metamodel.

In all, we see migration as a series of automated model transformations that are defined on metamodels to transform a source model into a target model using a distinct normalisation step. This approach is generic in the sense that it can be applied to any conforming source and target model without loss of generality. To actually implement this approach we require (normalised) source and target metamodels, normalisation rules, and transformation rules.

Although the approach is generic, our industrial case imposes some practical restrictions on the enabling technologies. Spurred by the fact that the existing architecture documentation contained source models (partly) in UML statecharts, we decided to implement the different steps of our migration approach using MDA technologies. In the MDA vision, software development is considered to be a series of model transformations. Similarly, we consider software migration as a series of model transformations. Starting from UML, technologies compatible with MDA offer convenient and off-the-shelf means to define and manipulate models. Furthermore, the MetaObject Facility[1] (MOF) can be used for the definition of metamodels.

[1]http://www.omg.org/mof (June 2007)

Finally, various model transformation languages are available to define transformations.

We defined all transformations in the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL). An advantage of ATL is its syntax, which is similar to that of the Object Constraint Language[1] (OCL). This allows people that have been working with the UML metamodel to understand and create transformation rules with relative ease. The actual ATL transformation engine relies on two implementations of MOF: the Eclipse Modeling Framework[2] (EMF) and the Metadata Repository[3] (MDR). The ATL transformation engine can be used in combination with MOF-based models and metamodels serialised with XML Metadata Interchange[4] (XMI). As our source metamodel we used the MOF-UML metamodel available from the Object Management Group[5] (OMG) [OMG, 2007a]. To create source models, we can simply use a UML modelling tool that supports XMI export. For the target metamodel we also used EMF as it allows for automatic generation of a primitive, tree-based editor for any arbitrary metamodel. This editor can then be used to inspect the results of our transformations.

7.5 Source Metamodel

In this chapter, we consider FSMs as the given starting point for the migration. The use of FSMs as a paradigm for supervisory control has been proposed by, for instance, Ramadge and Wonham [1987]. Here, the set of possible machine behaviours is considered to form a language. A discrete supervisory FSM is synthesised that restricts this language by disabling a subset of events to enforce valid machine behaviour. This requires the behaviour in all possible states for all requests to be specified explicitly using (un)conditional state transitions with associated triggers (events), and effects or state actions (manufacturing activities). When using this paradigm, concurrent execution is the result of independent parts of concurrently executing state machines that can optionally share events to synchronise. Consequently, multiple FSMs are used per controller (typically one for each type of request).

Our source models are specified using UML statechart diagrams. The relevant part of the metamodel is shown in Figure 7.4 . Apart from this metamodel, the UML specification also provides a large number of well-formedness rules, specified in OCL, of which a few are mentioned below.

[1]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)
[2]http://www.eclipse.org/emf (June 2007)
[3]http://mdr.netbeans.org (June 2007)
[4]http://www.omg.org/mda/specs.htm#XMI (June 2007)
[5]http://www.omg.org (June 2007)

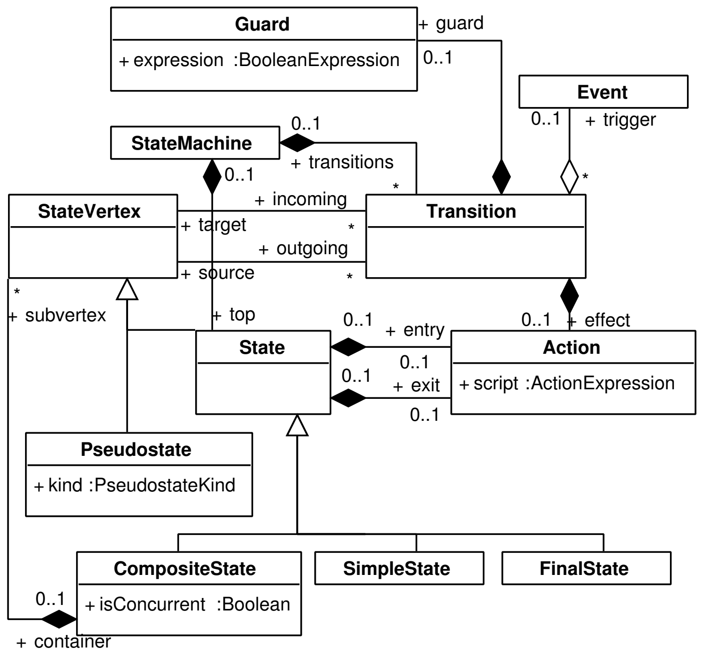


Figure 7.4: Source metamodel (excerpt from OMG [2007a])

Using this metamodel, UML state machines can be constructed that model behaviour as a traversal of a graph of state nodes interconnected by transition arcs.

In Figure 7.4 a state node, or StateVertex, is the target or source of any number of Transitions and can be of different types. A State represents a situation in which some invariant over state variables holds. In addition, an optional entry or exit Action is executed when the state is entered or exited. The metamodel defines different types of States. A CompositeState contains (owns) a number of substates (subvertex). If a CompositeState is concurrent (isConcurrent) it contains at least two composite substates that execute in parallel. A SimpleState is a State without any substates. Execution of an enclosing CompositeState ends when a FinalState is entered.

Next to state nodes that describe a distinct situation, the metamodel also offers a type of StateVertex that models a transient node of a state graph: a Pseudostate. It allows modelling of more complex (conditional) transition paths. Three types of pseudo-states (PseudostateKind) are relevant for the state models in this chapter: initial, choice, and junction. An initial Pseudostate is the default node of a CompositeState. A choice Pseudostate is used to create a dynamic conditional branch that depends on the action on its incoming transitions. Alternative paths may be joined using a junction Pseudostate.

Nodes in a state machine are connected by transitions that model the Transition from one State (source) to another (target). A Transition is fired by an Event (trigger). A Transition without such an explicit trigger is fired by an implicit completion Event that is generated upon completion of all activities in the currently active State. A Guard is a Boolean expression attached to a Transition that disables or enables its firing upon occurrence of its trigger (depending on whether it evaluates to true or to false). The effect of a Transition specifies an Action to be executed upon its firing. Note that, although there might be a causal relationship between actions and events (e.g., a call event generated by a call action), UML does not allow to make such a relationship explicit without the use of OCL. Finally, a StateMachine consists of a set of transitions and a top State that is a CompositeState.

As an example of how this metamodel is used in practice, consider the state machines in Figure 7.5 and 7.6 on page 142, which correspond to the process wafer and unload wafer requests as introduced in Section 7.3.2. Such state machines are the source models for the migration. Note that our example requests were adopted from two distinct supervisory control components with an indicative order of magnitude of 10 requests, 10-10^2 states, and 10^2-10^3 transitions. Although we use actual manufacturing requests as running examples, we do not depict or discuss these requests in full detail for reasons of confidentiality.

From the number of choice pseudo-states and guarded transitions it becomes clear that conditional execution is the dominant concern in the process wafer request in Figure 7.5 . In other words, the activated path is dependent on conditional synchronisation (e.g., wafer@measure) with other, concurrently executing requests.

Figure 7.6 on page 142 illustrates that after the actual transfer of the wafer (TRANSFER_FINISHED) the alternative completion sequences of subsequent activities, which are associated with the UR_moved and WS_moved events, are specified exhaustively. Furthermore, observe the use of two distinct resource usage patterns for WS and UR in our unload wafer request: for WS only an available Event (WS available) and release Action (release WS) are specified, for UR also a claim Action (claim UR) has been specified.

Note that for reasons of simplicity, we choose not to include resource usage and setups in the specification of the process wafer request. Even from our example requests it becomes clear that, in practice, concerns are addressed using a multitude of idioms and constructs. This is the main reason for the introduction of our normalisation step.

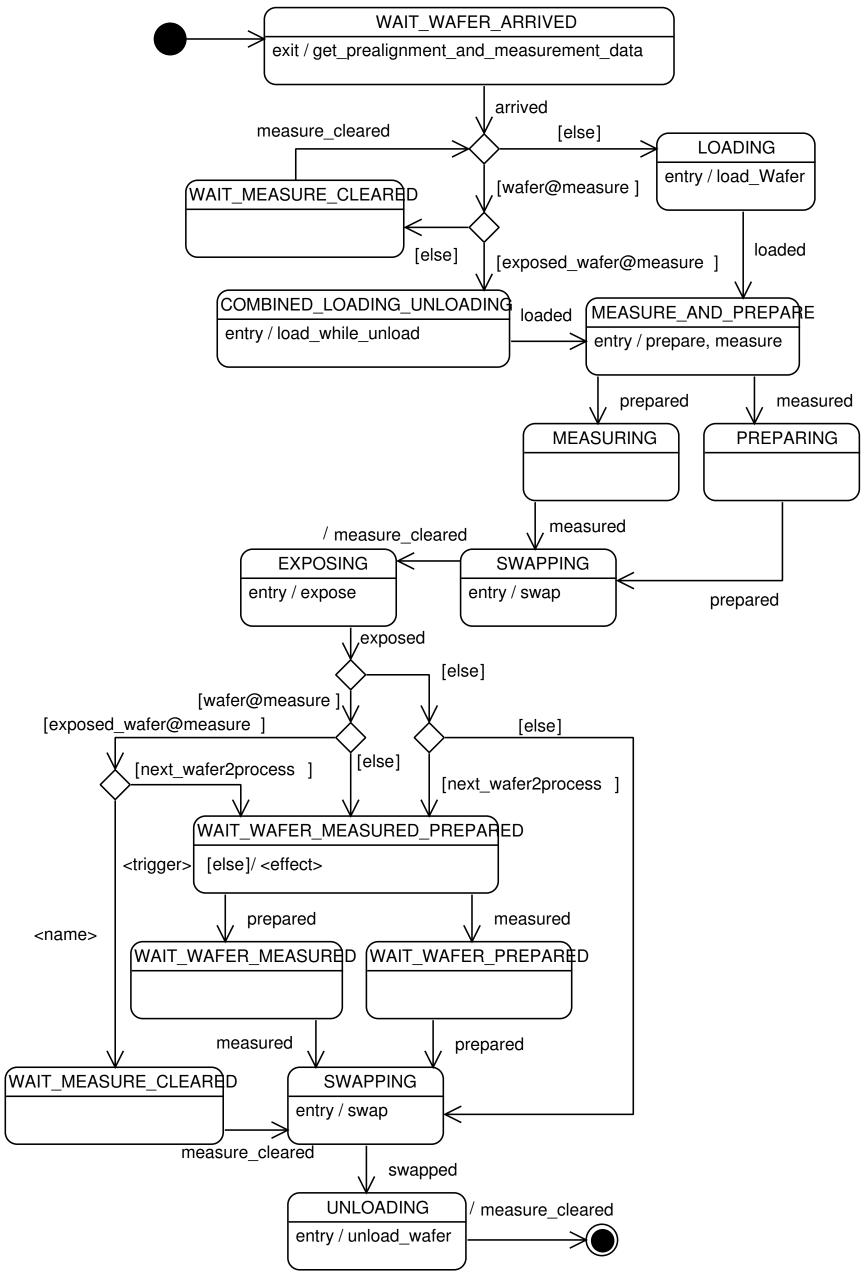


Figure 7.5: Process wafer request

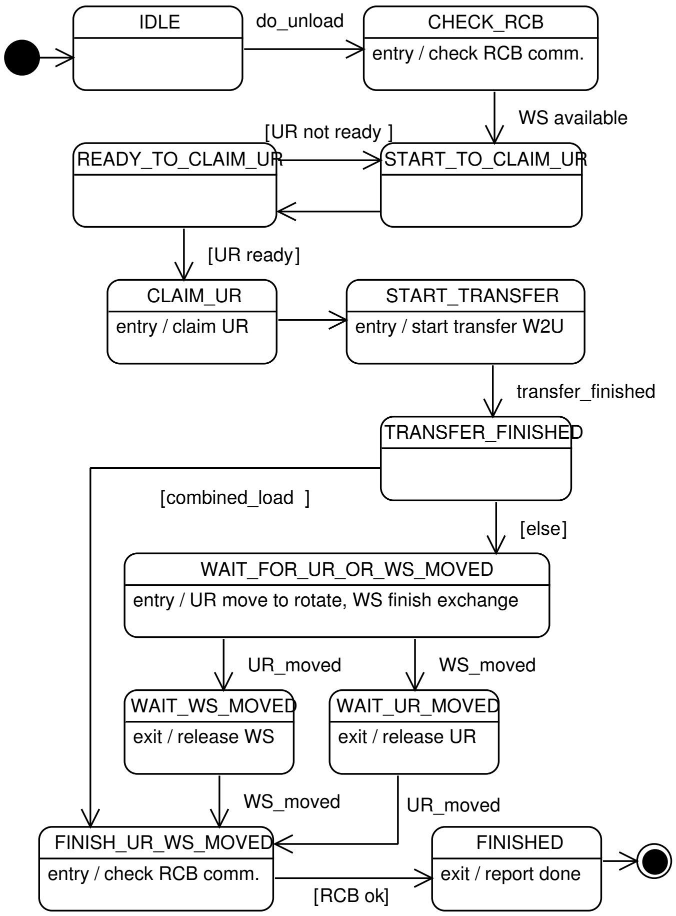


Figure 7.6: Unload wafer request

7.6 Normalisation Rules

UML, as a generic modelling language, lacks constructs to support its application in the domain of SMC systems. This makes that, when using 'plain' UML, various design idioms are available for handling SMC concerns. For instance, guards (e.g., 'subsystem is available') were often modelled as events (e.g., 'subsystem becomes available') although these are fundamentally different. Similarly, manufacturing activities can be specified as actions on state transitions or as actions in separate states. This idiom diversity is fuelled further by tool limitations. For instance, tools that support a specific UML version, do not necessarily support all of its constructs.

To define architecture transformations, we need source models in a normalised form. These normalised models are associated with a metamodel that adds constraints to the legacy source metamodel and augments it with SMC-specific constructs. These constraints and additional model elements are used in well-formedness rules that prescribe how to specify SMC-specific concerns. For this, UML allows attaching constraints to model elements using OCL, and for the definition of additional model elements by stereotypes. Together, these enable the definition of a suitable UML-SMC profile for the normalised source metamodel. Example diagrams that conform to this profile are shown in Figures 7.7 on page 148 and 7.8 on page 149.

Normalised source models have to comply to a set of well-formedness rules. Most importantly, concerns have to be specified in a uniform way. We have defined a standardised idiom for the concerns as identified in Section 7.3.3. We introduce this idiom by example of Figures 7.7 on page 148 and 7.8 on page 149. Normalisation involves modifying source models to remove any violation of these well-formedness rules. Note that, due to the diversity of the idioms used in the source models, normalisation is performed manually in our case study.

Table 7.1 on the following page lists the stereotypes that we define as part of the SMC profile. Next to stereotypes, the profile also defines a number of constraints. Listing 7.1 on the next page lists some of these constraints, specified in OCL as invariants over the UML metamodel (*C1*-*C4*). We merely use the constraints indicated by the `def` keyword to define extra properties on the elements mentioned in their `context`. This simplifies the specification of other constraints. Application of these stereotypes and constraints is discussed below.

Intuitively, the normalisation is context dependent and requires (some) domain knowledge. Moreover, the normalisation rules not only depend on the specific source paradigm but also on the modelling conventions as encountered in the specific (industrial) migration context. Therefore, we illustrate the normalisation step by defining the used context-specific normalisation rules for our case study.

Table 7.1: SMC profile stereotypes

Stereotype	baseClass	description
≪wait≫	State	wait for resource state
≪claim≫	Action	claim resource action
≪release≫	Action	release resources action
≪available≫	Guard	resource available guard
≪available≫	Event	resource becomes available event

```
context Action def:
  -- an action is a release action if a stereotype named 'release' is
    applied to it
  let isRelease : Boolean = self.stereotype->exists(s|s.name='release
   ')

context State def:
  -- a state is a wait state if a stereotype named 'wait' is applied
   to it
  let isWait : Boolean = self.stereotype->exists(s|s.name='wait')

context Event def:
  -- an event is an available event if a stereotype named 'available'
    is applied to it
  let isRelease : Boolean = self.stereotype->exists(s|s.name='release
   ')

-- C1: all release Actions are state exit actions
context Action inv:
  isRelease implies State.allInstances->exists(s|s.exit=self))

-- C2: a wait state has at least one outgoing transition triggered
 by an available event
context State inv:
  isWait implies outgoing->exists(t|t.trigger.isAvailable))

-- C3: state entry actions are actions that execute manufacturing
 activities (i.e., without stereotype)
context State inv:
  entry.stereotype->isEmpty

-- C4: all state nodes have no more than two incoming and outgoing
 transitions
context StateVertex inv:
  outgoing->size() <= 2 and incoming->size() <= 2
```

Listing 7.1: Some well-formedness rules of the SMC profile, in OCL

Subsystem setups In the source model, subsystem state consistency is ensured by specifying setup transitions for every possible subsystem state at design-time. In practice, this is not done exhaustively. Instead, domain-knowledge is used to limit the number of setup related alternative transitions. Although subsystem setups can be performed automatically using the TRS paradigm and, thus, do not need to be specified explicitly, we do preserve them during the normalisation step. This in fact ensures that the migrated control system mimics the behaviour of the legacy control system exactly. When reconsidering Figure 7.6 on page 142 and 7.8 on page 149, the move to rotate Action is in fact a resource setup.

For the normalised source model we do not use a specific idiom for setup activities; setups are modelled as any other manufacturing activity. If we would be less concerned with exact preservation of behaviour, setup activities could be simply removed during normalisation. In that case, domain knowledge is required to distinguish between setup activities and manufacturing activities.

Subsystem usage The pattern to address the 'subsystem usage' concern is best understood from one of the orthogonal regions in the composite state in Figure 7.8 on page 149. Before a manufacturing activity (e.g., finish exchange) that requires a certain subsystem (WS) is executed, a choice pseudo-state is entered. Then, if the required resource is available ([WS available]), it is claimed (claim WS) by the transition towards the state in which the manufacturing activity is executed (FINISH). Otherwise, a state (WAIT_FOR_WS) is entered that is only left when an event occurs indicating the resource has become available (WS available). The resource is claimed (claim WS) on the transition triggered by that event. Once the manufacturing activity is performed, claimed resources are released again by a release action that is executed when exiting the state (release). This pattern can easily be generalised.

We use the stereotypes defined by the SMC profile (Table 7.1) to distinguish between Actions, Guards, Events, and States related to the use of subsystems and those related to the execution of manufacturing activities (to which no stereotypes are applied). Normalisation introduces stereotypes for specific model elements that are related to the subsystem usage concern. Furthermore, from Figure 7.6 on page 142, and its normalised counterpart in Figure 7.8 on page 149, it can be seen that additional model elements are introduced to complete the pattern described above. Note that in Figures 7.7 on page 148 and 7.8 on page 149 stereotypes are displayed only for states: this is a limitation of the UML tool we are using (i.e., 'Poseidon for UML').

Application of the stereotypes to source models requires domain knowledge to recognise the subsystem usage concern. This becomes apparent when reconsidering Figure 7.8 on page 149. Here, WAIT_FOR_WS is a state in which the system waits for a subsystem to become available. This is intuitively different from the WAIT_WAFER_MEASURED states in Figure 7.7 on page 148, where the intention is to specify that the system waits for a manufacturing activity to be completed. The ≪wait≫ stereotype is only applied to the former state.

For normalisation of source models we require that resource usage patterns are made complete. In Figure 7.6 on page 142, for instance, only a release action is specified for WS. In Listing 7.1 on page 144 `C1`, `C2`, and `C3` are related to the subsystem usage pattern. `C1` specifies that a ≪release≫ Action only occurs as a state exit Action. `C2` states that at least one of the outgoing Transitions for a ≪wait≫ State is triggered by an ≪available≫ Event. Finally, to conform to constraint `C3`, all Actions related to manufacturing activities are moved to States as entry Actions. An example of this is the report done (entry) Action in Figure 7.6 on page 142 that was normalised to an exit Action (Figure 7.8 on page 149).

Synchronous execution Synchronisation between subsequent manufacturing activities in the source models is simply achieved by their order in the state machine. Furthermore, synchronisation between subsystem state transitions is not modelled at this level. As such, no specific idiom is used to specify this concern. In general, however, we have to take this concern into account while normalising the patterns associated with other concerns. While inserting and moving activities we have to make sure that we do not change their order in the normalised source model.

Concurrent execution In the original source models, concurrency was often modelled using States, including Actions that *start* two or more manufacturing activities and separate transition paths for all possible completion sequences, which are enabled by (external) completion Events. As an example, consider state MEASURE_AND_PREPARE and associated completions events prepared and measured in Figure 7.5 on page 141. Because those events can only be associated with their corresponding manufacturing activities using naming conventions, such an approach complicates the determination of the scope of concurrent execution. Therefore, we require that concurrency is modelled using a concurrent CompositeState containing (orthogonal) regions. This implies that during normalisation, manufacturing activities are mapped to CompositeStates when they are started in a single State node and alternative completion sequences are specified exhaustively. Figure 7.8 on page 149 contains an example of concurrent execution, where two resource usage patterns are executed in parallel.

Conditional execution The idiom for conditional execution is more complicated. First, we require it to be specified using a choice Pseudostate with two outgoing Transitions. One specifies some condition as a Guard; the other specifies [else] as a Guard. Furthermore, we require 'proper' nesting of conditional activation paths in a state machine. This means that we require pairs of corresponding, alternative paths through the state machine to be merged one at a time (using junction Pseudostates), and in reverse order. Figure 7.7 on the following page contains several (nested) examples of this pattern (choice and junction Pseudostates are depicted using diamonds and the smaller black circles, respectively).

Without this requirement for proper nesting, finding the set of States, and thus Actions, which are enabled when some Guard evaluates to true would become rather complicated. For the transformation of our source models to target models, finding this set of states is a necessary step. 'Non-proper' nesting occurs, for instance, in the bottom-half of the process wafer request in Figure 7.5 on page 141. This results in replication of the activities performed on each path during normalisation. The three CompositeStates in the bottom-half of Figure 7.7 on the following page illustrate this replication. Part of this particular normalisation step is covered by constraint *C4*, which states that a path through a state machine can only split in two paths and that no more than two paths can be joined in a single state node. Because the OCL constraint to express proper nesting is rather lengthy, we did not include it here.

7.7 Target Metamodel

We consider TRS as the given paradigm for the end-point of the migration. This end-point is based on a research prototype [Van den Nieuwelaar, 2004]. Using the TRS paradigm, a manufacturing request is translated into valid machine behaviour in two phases. First, upon arrival of a manufacturing request, a scheduling problem in the context of that request is instantiated during a planning phase. For this, the request is interpreted through rules that operate on capabilities (resource types) and behaviours (task types). Here, a manufacturing activity corresponds to a task and a mechatronic subsystem to a resource. The first phase results in a hierarchical digraph that consists of tasks and their (precedence) relations. Nodes in this graph can be composite to either denote a set of tasks that all need to be executed or to denote a set of tasks of which only one will be executed based on some condition. Second, a scheduling phase constructively assigns tasks in this digraph to specific resources over time [Viennot, 1986;

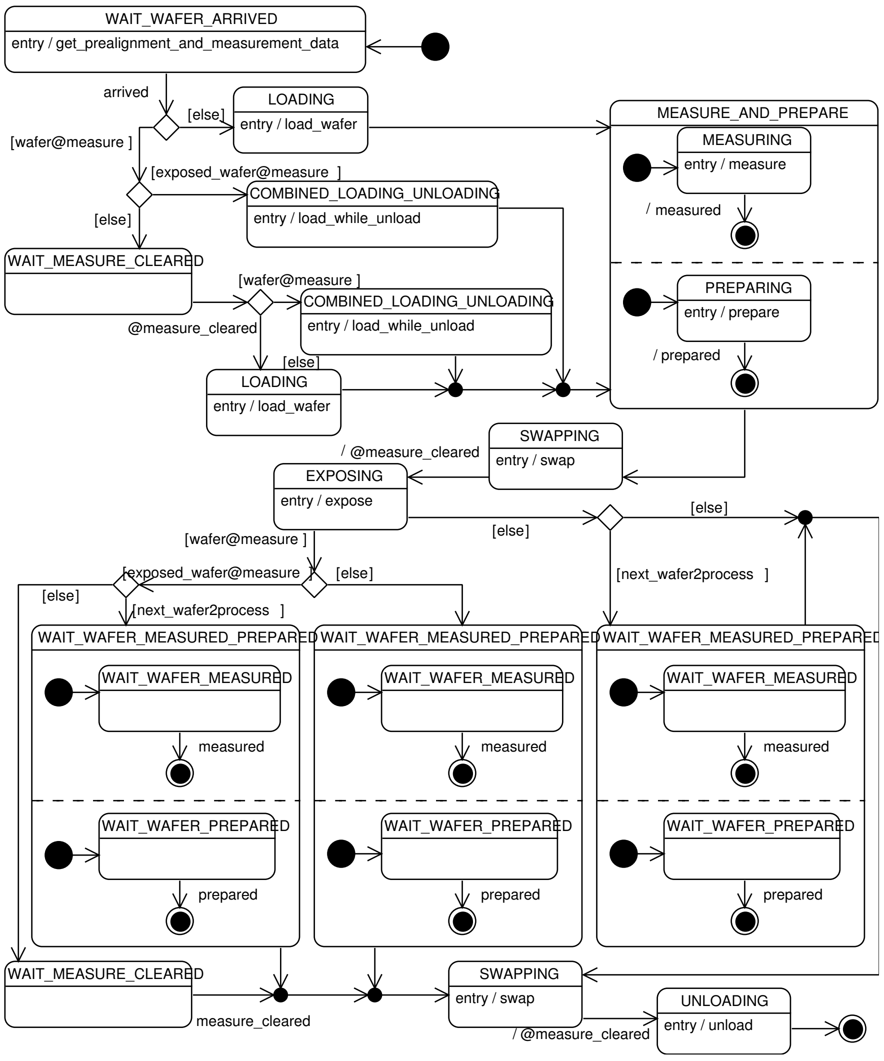


Figure 7.7: Normalised process wafer request

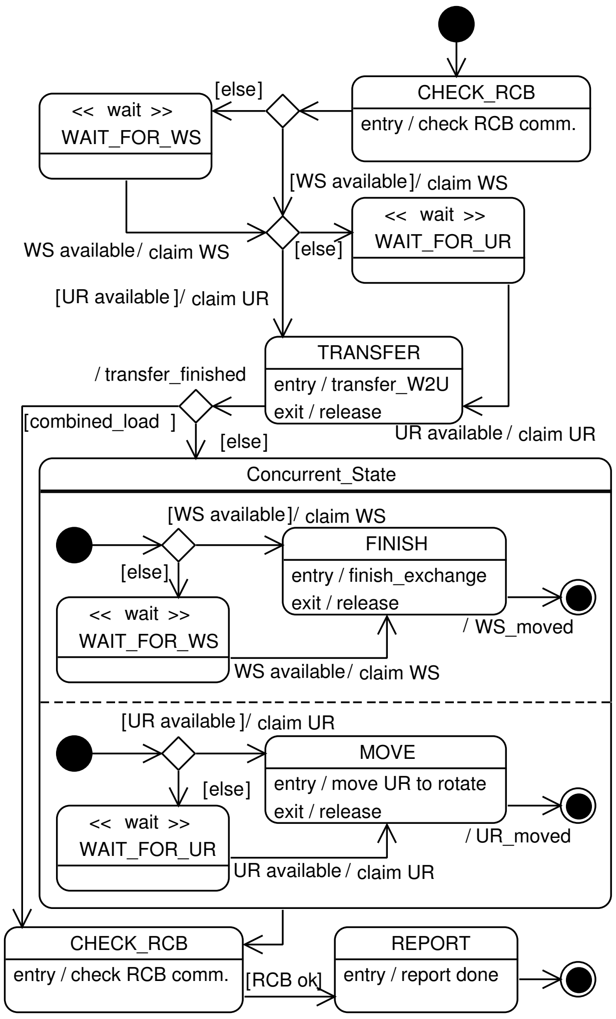


Figure 7.8: Normalised unload wafer request

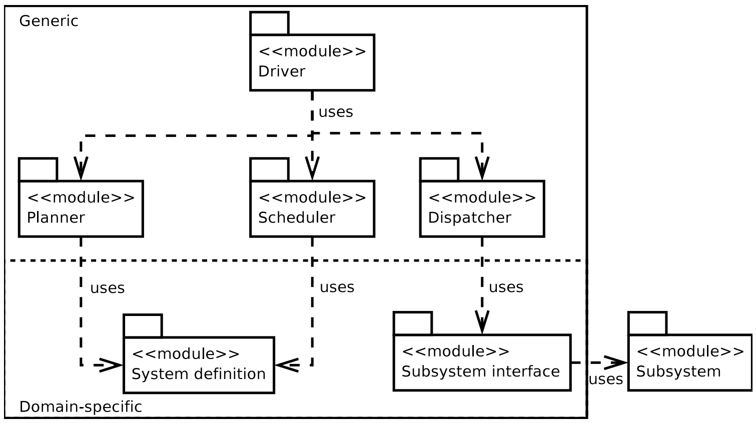


Figure 7.9: Module view for the product-line SMC architecture

Van den Nieuwelaar, 2004]. This results in a fully timed, coordinated TRS that can be dispatched for execution.

The end-point for our migration is a product-line architecture, of which Figure 7.9 displays the module view. In this architecture, the decisional responsibilities are assigned to three generic and reusable components: Planner, Scheduler, and Dispatcher. This product-line architecture offers variability with respect to tasks and resources and can be instantiated for a specific controller by implementing System definition and Subsystem interface modules. These modules define the specific system under control and implement the interfacing with lower-level components. The System definition module is amenable for code generation, allowing for a reduction of software development time and effort.

In order to define our target models, we introduce a governing target metamodel as depicted in Figure 7.10 . There, the system definition from Figure 7.9 is represented by the SystemDefinition, which serves as a root element. This system definition consists of a static and dynamic part. The static part defines the available Behaviours, Resources and Capabilities of the system under control. These are used to model types of manufacturing activities, subsystems, and types of subsystems. In addition, to address the subsystem usage concern, it defines which capabilities are required by which behaviour. Furthermore, the corresponding beginState and endState are specified in CapabilityUsage. These states are, for instance, used to determine sequence dependent setups.

The dynamic part of Figure 7.10 represents the rules for uniquely mapping a manufacturing Request to SimpleTasks, which are of a specific Behaviour, and assigning Resources that fulfil a required Capability. Every Task

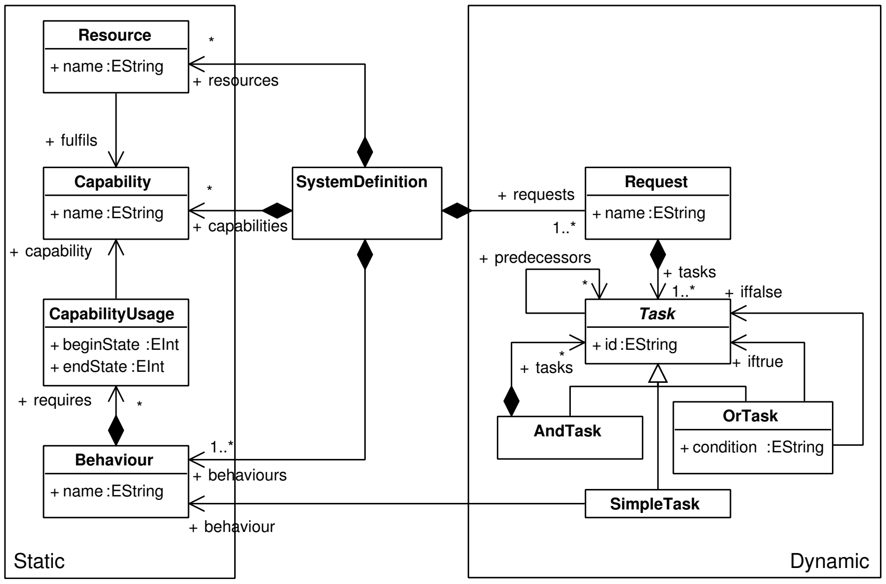


Figure 7.10: Target metamodel

includes a set of (direct) predecessors, that is, other Tasks that need to be executed before it can be dispatched. This relation is used to (dis)allow concurrency and imply synchronisation; in principle all tasks are executed in parallel, unless prevented by the predecessor relation. Conditional execution can be specified using OrTasks, that contain two Tasks (iftrue and iffalse) that may be composite. The evaluation of its condition determines which one will be dispatched. Finally, to cluster Tasks that *all* need to be performed, an AndTask can be used.

7.8 Transformation

Our transformation rules are defined as mappings from a normalised source metamodel (i.e., our UML profile) to a TRS metamodel. We used MOF to define the target metamodel rather than tailoring the UML using yet another profile. In this section we first introduce the transformation language that was used to define the transformation step of our migration approach.

For the definition of our transformations we used the following strategy. First, we indicate how elements in the normalised source metamodel are related to the primary elements of the target metamodel. Second, for each

```
rule Tasks {
 from s:UML!SimpleState (
   s.isTaskState and not thisModule.behaviourStates->includes(s))
 to t: TRS!SimpleTask (
   behaviour <- thisModule.resolveTemp(s.behaviourState,'b'),
   predecessors <- s.getPredecessors)
}
```

Listing 7.2: ATL example

of the identified SMC concerns we define and tailor transformation rules to relate the corresponding patterns in the normalised source model and the target model. These rules are described reasoning backwards, meaning that for each of the elements of the target metamodel we explain for what source model patterns they will be created.

In all, application of these transformation rules to a source model that conforms to our SMC profile, results in a target model that defines the System definition module for a particular SMC component (i.e., an instance of the architecture depicted in Figure 7.9 on page 150).

Next, we first introduce ATL. Then we discuss rules that generate the elements of the 'basic' types of the target metamodel in Figure 7.10 on the preceding page: SystemDefinition, Behaviour, Capability, Resource, Request, and SimpleTask. Subsequently, we describe rules to create the elements and relations related to the concerns as previously described in Section 7.3.3. Finally, we discuss the results of the application of these rules to our example requests.

7.8.1 The Atlas Transformation Language

All transformation rules are implemented using ATL. As an example, consider the ATL fragment in Listing 7.2. An ATL transformation module consists of rules that contain a **`from`** clause, specifying a source pattern (`s`), and a **`to`** clause specifying a target pattern (`t`). The source pattern consists of a source type (`UML!SimpleState`) and an optional guard, which is a Boolean expression specified in OCL. The target pattern consists of a set of elements that each specify a target type (`TRS!SimpleTask`) and an associated set of bindings. A binding refers to a feature of the type (e.g., `predecessors`) and specifies an expression that is used to initialise the feature. The source and target types in the transformation rules in this chapter refer to the source and target metamodels in Figures 7.4 on page 139 and 7.10 on the previous page. As such, the rule in Listing 7.2 matches SimpleStates that conform to some constraints expressed by the guard. This rule generates a SimpleTask for which it specifies a set of bindings.

For every element in the source model that matches the source pattern of a rule, the elements specified by the target pattern are created in the target model. Note that in ATL, the source model is read-only and the target model is write-only. This can also be seen from Listing 7.2 , where only the source model is navigated to initialise the features referred to in the bindings of the target pattern. Therefore, a specific value-resolution algorithm is used to initialise features: if the expression of a binding refers to another target element (created by the same rule) it is simply assigned, if it refers to a source element it is resolved by application of the rule that matches that source element and taking the default (first) target element.

For cases where the required target element is not the default element of another rule, ATL offers the 'resolveTemp' construct, a so-called helper operation. It takes a source model element and a reference to a specific target element of the matching rule as input parameters. In Listing 7.2 , for example, this is done in the binding of the `behaviour` feature. In this case `s.behaviourState` evaluates to a SimpleState that is matched by another rule with multiple target elements, of which the `'b'` target element is selected to bind to that feature.

Helpers are typically defined in the context of a metamodel element and effectively add a feature or operation to instances of that element (cf. the use of OCL definition constraints in Listing 7.1 on page 144). Alternatively, a helper can be defined without any context. Then, the default context of the complete transformation module, represented by the `thisModule` element, applies. The `resolveTemp` helper is also defined in this default context.

7.8.2 Basic Target Model Elements

SimpleTask and Behaviour SimpleTasks correspond to manufacturing activities, and Behaviours correspond to *types* of manufacturing activities in SMC systems. Therefore, to create SimpleTasks and Behaviours in the target model we need to identify Actions corresponding to manufacturing activities in the source model.

According to the UML SMC profile, an Action that corresponds to a manufacturing activity has no stereotype and is executed as a State entry Action (see *C3* in Listing 7.1 on page 144). For every such Action, a SimpleTask needs to be created. This is specified by the rule in Listing 7.3 on the next page. It contains a guard that uses the `behaviourStates` helper to only match SimpleStates that map to a Behaviour. Note that in the specification, we do not map Actions to SimpleTasks, but instead we map the SimpleStates in which they are executed to SimpleTasks. This does not affect our migration results since Actions corresponding to SimpleTasks are always State entry Actions (by constraint *C3* of the SMC profile).

```
rule Behaviours {
 from s: UML!SimpleState (
   thisModule.behaviourStates->includes(s))
 to t: TRS!SimpleTask (
   behaviour <- b,
   predecessors <- s.getPredecessors),
 b: TRS!Behaviour (
   name <- s.entry.script.body,
   requires <- s.incoming->collect(i|i.source)->iterate(s; ss:Set(
    UML!SimpleState) = Set {}|ss->union(s.getResourceClaims)))
}
```

Listing 7.3: Rule for tasks and behaviours

In the source model, the executed behaviour is specified in the Action's script attribute. Therefore, Actions with identical script attributes effectively define an Action *type* and should be mapped to the same Behaviour. To implement this, the `behaviourStates` helper first selects the set of SimpleStates corresponding to a SimpleTask (i.e., all SimpleStates with entry Actions without stereotype) and subsequently determines the set of Actions with unique Behaviours. For all Actions in this set, a SimpleTask and a Behaviour are created by the behaviour rule. Additionally, we have implemented a rule that creates a SimpleTask for all other SimpleStates with such entry actions.

For (Simple)Tasks, the `predecessors` attribute has to be set to the set of direct predecessor tasks. Furthermore, a Behaviour's `requires` attribute is set to a CapabilityUsage element. This is discussed in Section 7.8.3 for the related synchronous execution and subsystem usage concerns.

Resource and Capability To create Resources and Capabilities we identify mechatronic subsystems in the source models. However, in the FSM paradigm, subsystems are not modelled explicitly. Hence, the source model does not contain elements that directly correspond to Resources and Capabilities. We can, however, take advantage of the fact that in the FSM paradigm, subsystems are explicitly claimed. We create Resources in the target model based on Actions that claim a specific subsystem, that is, Actions to which the ≪claim≫ stereotype has been applied. Furthermore, for every resource we simply create a separate Capability (Resource type).

In the specification of the involved transformation rules (not shown) we had to take into account that Capabilities can be claimed multiple times during a single request. This results in multiple Actions claiming the same Capability. Because we do not want to create a separate Capability for each of the Actions claiming the same capability, we defined a helper similar to the behaviourStates helper.

SystemDefinition and Request The SystemDefinition root element in a target model contains all required elements that define the domain specific part of an SMC controller. As such, this element corresponds to a complete source model.

A Request encompasses rules that determine how that particular manufacturing request, such as our unload wafer from Figure 7.6 on page 142, is planned. Planning rules involve a set of Tasks and corresponding predecessor relations. Additionally, a Task can be an AndTask or an OrTask. In the source model, a complete state machine is used to specify how a manufacturing request is to be executed. So, we create a Request element in the target model for every StateMachine in the source model.

Listing 7.4 on the next page shows the ATL specification of this mapping. The `Request` rule generates a Request element for every StateMachine in the source model. This Request contains tasks which are created by other rules. As will be explained later, States or Guards in the source model may map to Tasks in the target model. Because the tasks in our target model may be composite in which case they *own* other tasks, we should take care not to select all model elements in the complete state machine that map to a Task. Instead, for a Request we discard all States or Guards inside a CompositeState other than the top, and on paths that are only conditionally enabled (i.e., by a transition's guard). To this end, we defined two additional generic helpers. First, `rootOfSubTree` takes a set of states as argument and recursively selects the 'first' state of that set (i.e., the one without incoming transitions from other states in the set). Second, `getTaskModelElements` is applied to that 'first' State to collect all model elements that map to a Task. In essence, this helper takes a set of states and traverses this set as a state 'tree' starting from the State (or Guard) it is applied to, and bypassing CompositeStates and conditional paths. During this traversal it collects all model elements it encounters that map to a Task (i.e., Guards or States).

The `SystemDefinition` rule generates a SystemDefinition element that corresponds to the complete source model. The `behaviours`, `resources` and `capability` features of the SystemDefinition element are bound to the result of other rules. In particular for `behaviours` and `resources` we had to use the `resolveTemp` helper as these are not created by the default target elements of the involved rules. In this case, the relevant source model elements are selected by two helpers that are defined in the context of the transformation module itself: `behaviourStates` gives all the source model elements (SimpleStates) that map to a Behaviour, and `resourceActions` gives all the source model elements (≪claim≫ Actions) that map to a Resource. The `request`feature is bound to the elements created by the `Request` rule for all StateMachines in the source model.

```
rule Request {
  from sm: UML!StateMachine
  to rq: TRS!Request (
    tasks <- thisModule.rootOfSubTree(sm.top.subvertex,sm.top.
     subvertex->asSequence()->first()).getTaskModelElements(sm.top.
     subvertex))
}
rule SystemDefinition {
  from sm: UML!Model
  to sd: TRS!SystemDefinition (
    behaviours <- thisModule.behaviourStates->collect(e|thisModule.
     resolveTemp(e,'b')),
    resources <- thisModule.claimActions->collect(e|thisModule.
     resolveTemp(e,'r')),
    capabilities <- thisModule.claimActions,
    requests <- UML!StateMachine->allInstances())
}
```

Listing 7.4: Rule for SystemDefinition and Requests

7.8.3 Concern-Based Transformation Rules

Resource usage To address the resource usage concern we need to relate Behaviours to the Resources and Capabilities (resource types) they require. In the target metamodel, CapabilityUsage elements are used to this end. However, we cannot derive the CapabilityUsage elements in the target model directly, since our source models only contain dynamic information. Consequently, we will have to derive them indirectly instead.

For each subsystem usage pattern, as described in Section 7.6 we conclude that the subsystems claimed at that point are required for the corresponding manufacturing activity. These are all the subsystems that are claimed after the previous release action. In the target model, CapabilityUsage elements are then defined connecting the corresponding Behaviour and Capabilities. For our unload wafer request, for instance, this results in the definition of a CapabilityUsage element relating the transfer W2U behaviour to the WS capability.

The **from** clause of the rule in Listing 7.5 matches all ≪wait≫ States, using the `isWait` attribute helper. The **to** clause of this rule creates a CapabilityUsage element in the target model. The `resolveTemp` helper is used to set the `capability` attribute to the target of the rule that matches the ≪claim≫ Action involved in the resource usage pattern. Next, a Behaviour is linked to CapabilityUsage elements by its `requires` feature. Listing 7.3 on page 154 shows that this is done by first selecting all States directly preceding the State in which an Action that corresponds to the Behaviour is executed. On each of these predecessor States, we iteratively

```
rule ResourceUsage {
 from s:UML!SimpleState (s.isWait)
 to cu: TRS!CapabilityUsage (
   capability <- thisModule.claimActions->select(a|a.script.body=s.
    outgoing->select(t|t.effect.isClaim).effect.script.body))
}
```

Listing 7.5: Rule for resource usage pattern

call the `getResourceClaims` helper that recursively finds all ≪wait≫ States by backwards traversal of the state machine until a ≪release≫ Action is encountered. A ≪release≫ Action releases all claimed subsystems. The ≪wait≫ States in the returned set match the ResourceUsage rule and the Behaviour is linked by its requires attribute to the corresponding CapabilityUsage elements.

Resource setups In the target model, setups are automatically inserted by the generic (solving) part of the product-line architecture. This is done at run-time, based on mismatching beginState and endState attributes of the CapabilityUsage element. To some extent, these could be derived from the explicitly specified setups in the source model.

In this chapter, however, we do not define a corresponding transformation rule as it depends heavily on domain knowledge. Using our transformations, setups will explicitly end up in the target model as just another task and behaviour. As said, this ensures that the migrated control system mimics the behaviour of the legacy control system exactly, thus resulting in a validated and acceptable baseline.

Synchronous execution The target model defines precedence relations between those Tasks that require synchronisation (within the same Request). In principle, these relations follow from the execution order of the manufacturing activities and the corresponding Actions within a normalised state machine. In addition, (virtual) resources can be used for external synchronisation.

For synchronisation within a Request, predecessor relations are created for every task by searching for its set of (direct) predecessor tasks. For this we have defined two helpers that both operate on the elements that match rules that create Tasks. The first helper is depicted in Listing 7.6 on the following page and is defined on StateVertex whereas the second one is defined on Guard. For each Task, one of these `getPredecessors` helpers is invoked on its corresponding StateVertex or Guard. These helpers determine whether the current element (`self`) corresponds to a task. If so, this element is returned. Otherwise, the helper is recursively applied to

```
helper context UML!StateVertex def: getPredecessors:Set(UML!
 ModelElement) =
  if self.incoming->isEmpty() then
    Set{}
  else if self.isOrTaskStateJoin then
    self.getFork.outgoing->collect(e|e.guard)->select(e|e.
     isOrTaskGuard)
  else if self.incoming->collect(e|e.guard)->select(e|not e.
   oclIsUndefined())->exists(e|e.isOrTaskGuard or if e.oppositeGuard.
   oclIsUndefined() then false else e.oppositeGuard.isOrTaskGuard
   endif) then
    Set{}
  else if self.incoming->collect(e|e.source)->select(e|e.isTaskState)
   ->isEmpty() then
    self.incoming->collect(e|e.source.getPredecessors)->flatten()
  else
    self.incoming->collect(e|e.source)->select(e|e.isTaskState)
  endif endif endif endif;
```

Listing 7.6: Collect predecessors on StateVertex

the finite set of all direct preceding modelling elements that may map to a task.

Concurrent execution The normalised pattern for concurrency, as discussed in Section 7.6, is a CompositeState with orthogonal regions. To address the concurrent execution concern we need to identify instances of such patterns in the source model.

We defined a transformation rule that creates an AndTask for every concurrent CompositeState in the source model except for the top CompositeState of the StateMachine. Basically, the predecessors relation is the mechanism used in the target model to (dis)allow concurrency: if two tasks are not related by the transitive closure of the predecessors relation, they can execute concurrently. Now, these potentially concurrent tasks are executed as soon as execution of their predecessors has finished and the required resources are available. In turn, this also implies that a task can have multiple (concurrent) predecessors. Collecting predecessor tasks was already discussed in the previous paragraph.

Conditional Execution As discussed in Section 7.6, the normalised source model uses a state with two outgoing guarded transitions to specify conditional execution. Every two alternative conditional branches in a source model are mapped to an OrTask in the target model. This OrTask contains two subtasks (iftrue and iffalse), which may be composite and represent the two conditionally executed branches following a State with two outgoing

```
rule ConditionalExecution {
 from g:UML!Guard (g.isOrTaskGuard)
 to t: TRS!OrTask(
   condition <- g.expression.body,
   iftrue <- at_true,
   iffalse <- at_false,
   predecessors <- g.getPredecessors),
 at_true: TRS!AndTask(
   tasks <- g.transition.target.getTaskModelElements(g.
    guardedTaskStates))
 at_false: TRS!AndTask(
   tasks <- g.oppositeGuard.transition.target.getTaskModelElements(g
    .oppositeGuard.guardedTaskStates)
}
```

Listing 7.7: Rule for conditional execution patterns

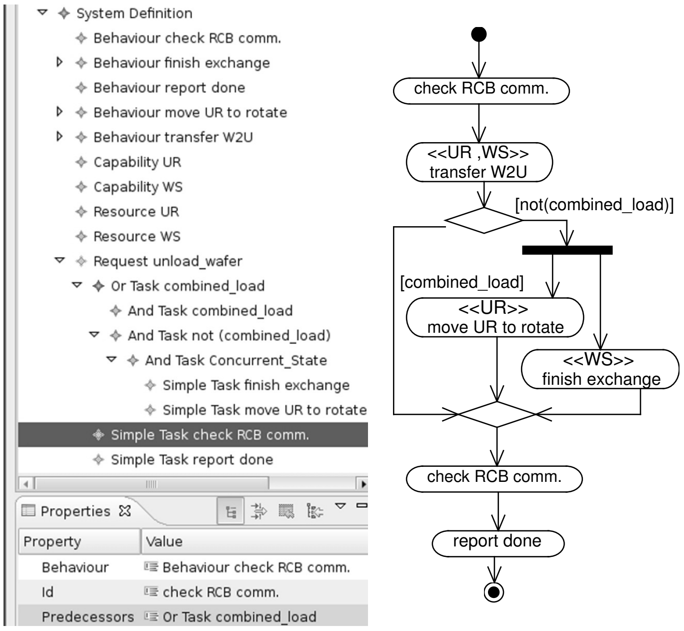
guarded Transitions. Subsequently, for the creation of those subtasks, we need to find all model elements that map to a task in each of the branches.

The specification of this transformation rule is depicted in Listing 7.7. This rule matches one of the Guards (not the else) for every conditional execution pattern, determined by the `isOrTaskGuard` helper. It creates an AndTask for each of the two branches using the `getTaskModelElements` helper. The set of States that this helper uses to determine the scope in which it has to select all ModelElements that map to a Task is calculated by the `guardedTaskStates` helper. This helper selects all States ‘guarded’ by some guard. To this end, it calculates the difference between the path through the state machine that starts from the target of the conditional transition and the corresponding alternative transition path.

7.8.4 Transformation Results

In total, we needed approximately 300 lines of ATL code to implement all the necessary transformation rules and helpers for the transformation step of our migration approach. Once the source model, source metamodel, target metamodel, and transformation module are defined and located, the ATL transformation engine generates the target model (e.g., a system definition) in its serialised form. The results as obtained for the normalised unload wafer request are depicted in Figure 7.11 on the following page.

Figure 7.11(a) on the next page shows a screen capture of the created TRS target model, inspected using the tree-based editor that was generated for our TRS metamodel by the EMF plugin. There, TRS model elements are shown in a tree structure to indicate containment. Furthermore, it can be seen that we are dealing with an SMC component that accepts a Request



System Definition
- Behaviour check RCB comm.
- Behaviour finish exchange
- Behaviour report done
- Behaviour move UR to rotate
- Behaviour transfer W2U
- Capability UR
- Capability WS
- Resource UR
- Resource WS
- Request unload_wafer
  - Or Task combined_load
    - And Task combined_load
    - And Task not (combined_load)
      - And Task Concurrent_State
        - Simple Task finish exchange
        - Simple Task move UR to rotate
  - Simple Task check RCB comm.
  - Simple Task report done

Properties

Property	Value
Behaviour	Behaviour check RCB comm.
Id	check RCB comm.
Predecessors	Or Task combined_load

(a) TRS target model **(b)** Activity Diagram

Figure 7.11: Results for unload wafer request

unload_wafer. The selected element under the Properties tab in the bottom part reveals that "SimpleTask check RCB comm." can only be dispatched after its predecessor "OrTask combined_load" has been executed.

The consequence of using a custom metamodel is that we only have the basic generated editor to visualise and document our transformation results. Again, we turned to model transformations to solve this problem. As there is no suitable graphical representation for complete TRS models yet, we defined a transformation that maps a TRS model to UML Activity Graphs for the dynamic part (one for each request) and a UML Class model for the static part. The result of this transformation can easily be displayed using UML tools. Figure 7.11(b) on this page, for instance, shows the dynamic part of our unload wafer request displayed as an UML Activity Graph.

Note that, we merely use UML notation to *represent* part of the task resource model. As such, the semantics are not identical to that of UML activity graphs, but only similar. We represent Tasks as Activities stereotyped with the resource they require. The transition represent predecessor relationships (in reverse direction). For AndTasks we use fork Pseudostates (represented by the horizontal black bar). A complete AndTask is thus represented by the subgraph that starts with a fork and ends when the two concurrent paths are joined. OrTasks are represented using choice Pseudostates (represented by a diamond with two outgoing arrows). Similar to the AndTask a complete OrTask is thus represented by the subgraph that starts with a choice Pseudostate and ends when the two conditional paths are joined. For convenience we did not explicitly represented the join of the two concurrent paths (i.e., using another horizontal bar); they are joined in the same node (the diamond with three incoming arrows) as the conditional paths.

7.9 Evaluation

Applicability Application of our generic, model-driven migration approach requires that the source view and target view can be defined using a metamodel. When this is possible, the actual migration from source to target constitutes a series of model transformations.

In practice, models are only made as complete and accurate as is demanded by their application. However, these demands become more stringent when these models are used as input for automated processing such as model transformations. As a result, the context-specific normalisation step is crucial to the applicability of our migration approach in industrial contexts where (source) models are typically used for communication and documentation purposes only.

MOF-based metamodels only provide the abstract syntax for conforming models and do not define how to visualise them (concrete syntax). In fact this is a drawback of using a custom metamodel: no model editors and viewers are available, apart from the basic editor as generated by the EMF plugin. In this chapter we again turned to model transformations to document and visualise our results. The use of model transformations provides an elegant and flexible way of generating architecture documentation that can easily be tailored to meet specific documentation requirements of a migration context. This is further discussed in Chapter 8.

It turned out that a model-driven migration approach based on MDA is useful for rapid (incremental) development of normalisation rules and transformation rules. That is, results can easily be visualised and documented given the wide variety of available tools.

Scalability With respect to the scalability of our approach we can safely state that our experiments are of the same order of magnitude as full-fledged component migrations for real-world wafer scanner applications. More concretely, the two requests that were migrated as a proof of concept account for approximately 10-20% of the source code for our SMC components. The application of our transformation rules to the two representative examples presented in this chapter requires less than 10 seconds to complete on a 1.7 GHz notebook. Furthermore, we expect the execution time to be linear with respect to the number of requests. More important for the execution time is the nesting depth of conditional paths. For our industrial case we have not encountered requests with deeper nesting than our example requests.

Effectiveness Our model-driven approach requires that implicit design decisions and design knowledge is consolidated and made explicit for the definition of metamodels and transformation rules. As such, the application of our approach to the SMC components of our case study increased the general understanding of concerns and the associated implications (and difficulties) surrounding the architecture migration of SMC systems. Moreover, the need for experts on both the domain and the target paradigm was confined to the definition of the normalisation and transformation rules.

The effectiveness of both the MDA approach and our model-driven migration approach depends partially on the ability of modelling, transformation and code generation tools to cooperate. As such, standards involved with the MDA, such as MOF, UML, and particularly XMI, play an important role. In practice, the availability of different versions of these specifications made it difficult to setup an appropriate tool chain. For instance, we could not use the latest version of our UML modelling tool (i.e., 'Poseidon for UML') because the UML metamodel it uses, was incompatible with the ATL transformation engine. Although we took the liberty of selecting tools that were able to cooperate, we still needed to implement some additional transformations using Extensible Stylesheet Language Transformations[1] (XSLT) to overcome some incompatibilities between the various tools. In industry it will not always be possible to select a specific set of tools for the migration given practical considerations such as licensing, support, and training costs.

Apart from tool support, the required human intervention during the normalisation step also determines the effectiveness of our migration approach. The complexity of the normalisation step depends on the number of constraints that the restricted source metamodel adds to the legacy source metamodel (if present). Here, a trade-off applies: fewer constraints make the transformation, which is typically automated, more complex because more specification alternatives have to be covered. For instance, if we would allow Actions corresponding to manufacturing activities to occur as Actions on Transitions, searching for predecessors would become much

[1]http://www.w3.org/TR/xslt (June 2007)

more complicated. On the other hand, the normalisation step requires less effort in that case.

In our case, the target metamodel specifies the domain-specific part of a product-line. We believe that model transformations are particularly applicable as a migration approach for the recurring migration of individual product-line members. In general, a model-based migration approach is beneficial in situations where a number of similar artefacts need to be migrated. Such a setting provides sufficient return on investment for the definition of metamodels, normalisation rules, and transformation rules.

More specifically, when considering the previously mentioned trade-off, a larger number of artefacts that need to be migrated justifies a higher investment in the definition of transformation rules, allowing for a less involved normalisation step. As another example of this trade-off, consider our assumption of proper nesting. It implies that alternative branches in a state machine are joined two at a time and in reverse order. One could relax this assumption (constraint) and implement a more intricate transformation rule to handle this relaxation.

Extensibility Currently, our transformation rules do not handle synchronisation across different requests. This could prove to be a limitation for the large scale application of our transformation rules. To this end, we would have to (at least) extend our profile to include a special type of Event to denote external events for such inter-request dependencies.

The overall extensibility of our migration approach is demonstrated by using source models with two distinct origins for our experiments. In the case of the unload wafer request we used the available architecture documentation of the involved SMC component. This documentation contained UML statechart diagrams for the component's requests, including our example request.

However, for the SMC component that performs the process wafer request, documentation was not available. We had to reconstruct the source model from the source code. For this we took advantage of the fact that this component was based on a proprietary library for FSMs. Using this library, the component implemented three concurrent state machines that covered the behaviour of all requests and combinations hereof. Figure 7.12 on the next page depicts one of the component's three state machines.

This particular state machine illustrates the typical result of an evolving software architecture: two legacy state-based components were augmented with a new supervisor. This supervisor was obtained by taking the product of the two legacy state-machines and adding two choice pseudostates (i.e., s1 and s11) to allow for different activation paths, based on legacy request combinations.

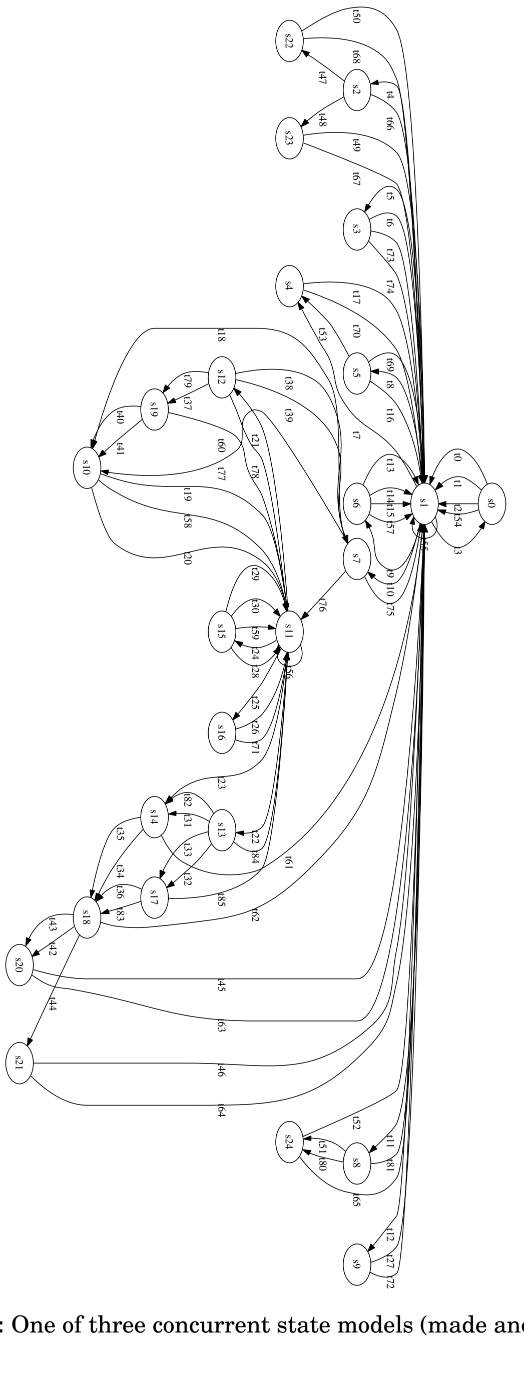


Figure 7.12: One of three concurrent state models (made anonymous)

We extracted the process wafer request state machine from the three implemented concurrent state machines and the corresponding source code by isolating state transition paths and combining them into a request state model. The resulting extracted source models were used as the input for our normalisation step. In fact, such an extraction step in which we isolate request state machines (i.e., to obtain models to be normalised) can be seen as an extension of the 'front-end' of our approach.

The 'back-end' of our approach can be extended as well by steps that further process the result of our model transformations. We already mentioned the generation of documentation. Another possible extension is the generation of source code to actually generate the System Definition module of the product-line architecture (Figure 7.9 on page 150). Both can be specified using model transformations.

Note that we did not yet consider the domain specific interface modules of the product-line architecture. However, this only constitutes a minor hurdle since we can simply encapsulate the existing source code bodies for each behavior (preserving interface functionality and behavior).

7.10 Conclusions and Future Work

In this chapter we formulated the migration of SMC systems as a model transformation problem. The starting point is an SMC architecture based on FSMs; the end point is a product-line SMC architecture based on TRSs. Our approach supports the generic migration of the product-line members.

We demonstrated that the development framework for the MDA can be successfully applied in a migration context as well: migration can be seen as a series of model transformations. We proposed a generic two-phased, model-driven migration approach that uses distinct normalisation and transformation steps to derive the modules required to instantiate the TRS product-line architecture for a particular (sub)system. The normalisation step is crucial in overcoming semi-formal, incomplete and ambiguous specifications as well as tool and language limitations. This normalisation step requires domain knowledge and manual effort, but makes our approach suited for industrial application.

A trade-off has been identified between the inherent complexity of automated transformations and the required manual effort during normalisation. Based on SMC-specific concerns and a normalised source metamodel, we have defined and implemented a set of generic transformation rules that support a migration towards TRS-based product-line architectures. The applicability of these rules has been illustrated for a real-world industrial case. Since our transformation rules operate on normalisations, they can be applied to FSM-TRS migrations of SMC systems without loss of generality.

The industrial case that motivated this chapter imposes not only the source and target paradigms but places practical constraints on the enabling technologies as well. Starting from UML, we selected technologies compatible with the MDA to setup a convenient tool-chain that supports the definition and manipulation of models. Using this tool chain, several requests from different SMC components have been migrated as a proof of concept. The experiences we gained from this exercise indicate that the application of model transformations not only increases the understandability of such a migration, but also reduces the need for domain experts.

As such, the main contributions of this chapter are:

- The illustrated applicability of the MDA approach to architecture migrations. To this end, we introduced a vital normalisation step that enables migrations in an industrial setting.
- A practical view on the use of metamodels and profiles for migrations in general and, more specifically, on the normalisation, and transformation of SMC source models.
- The specification of a set of model transformation rules, an SMC UML profile, and a TRS metamodel that can be applied to FSM-TRS migrations of SMC architectures.

We are in the process of extending our work along the following lines. First, we want to further investigate the extraction of source models for our transformation directly from source code. This may also enable (partial) formalisation and automation of our normalisation step. Second, at the other end of the migration, we want to extend our approach with code generation from TRS models for the application-specific modules of the TRS product-line architecture, again using technologies related to the MDA. This would provide for a full-fledged model-driven migration approach: from legacy code to new code through a series of model transformations.

Chapter 8

Visualisation of Domain-Specific Modelling Languages Using UML[1]

Currently, general-purpose modelling tools are often only used to draw diagrams for the purpose of documentation. The introduction of model-driven software development approaches involves the definition of domain-specific modelling languages that allow code generation. Although graphical representations of the involved models are important for documentation, the development of required visualisations and editors is cumbersome. In this chapter we propose to extend the typical model-driven approach with the automatic generation of diagrams for documentation. We illustrate the approach using the Model Driven Architecture in the domains of software architecture and control systems.

8.1 Introduction

Model-driven engineering refers to software development approaches in which models are considered the primary development artefacts [Bézivin, 2005] (instead of source code). In these approaches software models are gradually transformed (automatically) into source code by means of model transformations. Additionally, such models are used for other (automated) software engineering tasks, such as performance analysis.

Typically, model-driven engineering (MDE) approaches are based on modelling languages that offer abstractions focused on a particular domain. Such languages are referred to as domain-specific modelling languages

[1]This chapter was published earlier as: Graaf, Bas and Arie van Deursen. Visualisation of domain-specific modelling languages using UML. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2007)*, pages 586–595. IEEE Computer Society, 2007c

(DSMLs). From DSML models code is generated for a particular software platform. DSMLs have been developed for various types of domains, such as software engineering (e.g., software architecture [Medvidovic and Taylor, 1997]) and application domains (e.g., insurance products [Doyle et al., 2006]).

In general, the use of DSMLs has clear advantages over the use of general-purpose languages (GPLs) [Van Deursen and Klint, 1998]. More in particular, in the context of MDE, our experience in industrial case studies (see Chapters 5 and 7) indicates that the use of a GPL, such as the Unified Modeling Language[1] (UML), leads to (unnecessary) complex model transformations, for instance to generate code. As such, the introduction of MDE typically requires the development of DSMLs.

Although mechanisms are available to define and implement the abstract syntax of DSMLs, such as the MetaObject Facility[2] (MOF) and the Eclipse Modeling Framework[3] (EMF), not much support is available for the definition of their graphical notation (concrete syntax). As a result development of adequate graphical editors and visualisations requires considerable effort.

For some software engineering tasks, such editors are not required. For instance, developers can use a textual syntax for the creation of models that can subsequently be processed by model transformation tools. However, other tasks, such as documentation, do require some form of graphical representation. It is this problem that motivates this chapter.

The basic idea of this chapter is simple: when devising a new DSML we try to leverage existing visual notations and modelling tools. We propose to expand the typical MDE process in which abstract models are gradually transformed into code, with (partial) generation of documentation. To this end we combine the use of DSMLs for code generation and other automated software engineering tasks, with the use of UML for documentation. The approach uses model transformations to specify the mapping between DSMLs and UML. The diagrams corresponding to the resulting UML models, as visualised by off-the-shelf UML tools, are used in the documentation. To investigate the arguments for and against this idea, we study how

- this approach works for various architectural views;
- UML can be used as the target language for visualising these views; and
- model transformations can be used to specify and automate the mapping.

[1]http://www.uml.org (June 2007)
[2]http://www.omg.org/mof (June 2007)
[3]http://www.eclipse.org/emf (June 2007)

In practice, the extra effort required for the development of graphical editors can hamper the introduction of MDE. Consider the following scenario. A software development organisation decides to introduce MDE. Currently, the developers use UML. However, as in many other organisations, they only use UML modelling tools for *drawing* diagrams [Lange et al., 2006]. These diagrams are important for the communication with other stakeholders, as they constitute an essential part of the documentation. The introduction of MDE involves the definition of DSMLs from which code will be generated. Furthermore, as the developers are comfortable with using a textual syntax for these DSMLs, no graphical editors are developed. The result is that they now have to create DSML models for code generation as well as UML diagrams for documentation. Considering the current use of UML, as investigated by Lange et al. [2006], and the upcoming of MDE approaches, such as the Model Driven Architecture[1] (MDA), this is not an unlikely scenario.

We investigate the feasibility of our approach in the domain of software architecture. In Section 8.2 we introduce the languages specific to this domain, and the standard documentation approach. Our approach for the model-driven documentation of software architecture, MDAV, is presented in Section 8.3 and we report on a small case study in Section 8.4. The approach is easily applied to other domains as well. An additional (industrial) case study involving a different type of models is presented in Section 8.5. We discuss the benefits and limitations of the approach in Section 8.6. After discussing some related work in Section 8.7, we conclude with an overview of our contributions and opportunities for future work in Section 8.8.

8.2 Background

In this section we introduce modelling and documentation in the domain of software architecture. Furthermore, we discuss some of the technologies that enable our approach.

8.2.1 Software Architecture

Modelling Several notations have been developed to specify architectural models. These architecture description languages (ADLs) (see Medvidovic and Taylor [1997] for an overview) mostly consider an architecture to be a configuration of runtime components and connectors.

Due to their formal syntax and semantics ADLs enable automatic code generation and analysis. Despite these benefits, and although ADLs have

[1]http://www.omg.org/mda (June 2007)

received much attention from the architecture research community, they have not been applied much in industry [Kruchten et al., 2006].

Although UML is aimed at object-oriented modelling, it allows practitioners to address a wide range of issues [Medvidovic et al., 2002]. Therefore, and because of the availability of supporting (graphical) modelling tools, it is often used in practice to describe software architectures (e.g., see Chapter 3 and Lange et al. [2006]).

A drawback of using UML for this purpose is the semantic mismatch between architectural concepts and UML's concepts, which are aimed at object-oriented design. This results in compromises between completeness and legibility [Garlan et al., 2002]. Furthermore, for automatic processing of models (e.g., for code generation) the complexity of UML results in complex model transformations (see Chapters 5 and 7).

Documentation Because in industrial practice a software architecture is too complex to describe in a single stroke, different views are used for its documentation. Different types of views have been defined to address specific concerns. The two most prominent categories of views are module views and component-and-connector (C&C) views [Clements et al., 2002a].

A module view addresses the question of how a system is *developed*; it defines the most important implementation units (modules) and their relations. Module views are used, for instance, to evaluate the maintainability of a system as implied by its architecture.

A component-and-connector (C&C) view, on the other hand, addresses the question of how a system *works*. It describes a system in terms of runtime components and connectors. A component is an abstraction of a computational element; a connector is an abstraction of the way components interact. As such, a C&C view is more suited for analysis of runtime properties, such as performance.

More specific types of views are defined by imposing restrictions on the type of elements and relations allowed in a view. In a module-uses view, for instance, only 'uses' relations are allowed.

In the terminology of IEEE Std 1471-2000 [IEEE-1471, 2000], a view conforms to a viewpoint that "specifies the conventions for using and constructing a view". A viewpoint addresses a set of stakeholder concerns. A number of viewpoint sets is available from literature, such as [Clements et al., 2002a]. Furthermore, in practice also custom viewpoints are defined. Typically, a viewpoint definition prescribes a modelling language or notation that enables the specification of an architectural model that addresses the concerns of the viewpoint. As an example, a C&C viewpoint might refer to a particular ADL. In summary, a viewpoint defines a type of views and a view is a particular representation of a particular system.

In practice the architectural model for a view is primarily used as a figure or diagram (the view's primary presentation [Clements et al., 2002a]) in a document that describes the view. Because of their wide-acceptance and available tool support often UML diagrams are used for this (see Chapter 3).

8.2.2 Enabling MDE Technologies

Our approach for model-driven documentation is based on model transformations. This requires capabilities for (meta)modelling, model transformation, and model interchange.

For the definition of metamodels we use the MetaObject Facility[1] (MOF) and its implementation as an Eclipse plugin: the Eclipse Modeling Framework[2] (EMF).

The EMF plugin generates an implementation for a metamodel as a set of Java classes that offers an interface that allows developers to manipulate conforming models. These models can be serialised to a document in the Extensible Markup Language[3] (XML) using XML Metadata Interchange[4] (XMI). Additionally, a simple tree-based editor is generated that can be used as an Eclipse plugin for the creation and inspection of associated models. As an example consider the screenshot of such an editor in Figure 8.3(b) on page 175. This editor is also capable of validating a model against its metamodel.

The Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL) is based on EMF. We use it to define model transformations that are executed by a transformation engine. In ATL, transformations are defined in modules that consist of declarative transformation rules and helper operations. Using a syntax similar to that of the Object Constraint Language[5] (OCL), the transformation rules match model elements in a source model and create elements in a target model. A helper is defined in the context of a metamodel element, to which it effectively adds a feature.

For their input, model transformation tools typically use XMI serialisations of MOF-based (meta)models. In the case of UML, these models can simply be created and visualised using standard UML tooling.

[1]http://www.omg.org/mof (June 2007)
[2]http://www.eclipse.org/emf (June 2007)
[3]http://www.w3.org/XML (June 2007)
[4]http://www.omg.org/mda/specs.htm#XMI (June 2007)
[5]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)

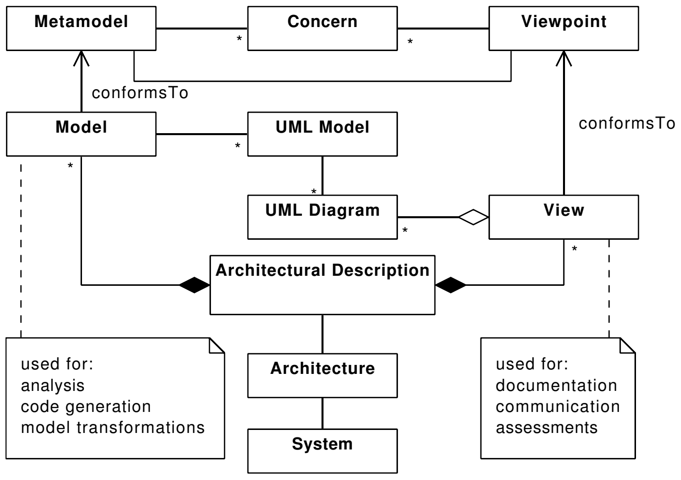


Figure 8.1: MDAV framework

8.3 Model-Driven Architectural Views

To take advantage of the power of DSMLs for code generation and other automated software engineering tasks and that of UML for documentation, we explicitly distinguish architectural documentation and architectural models. We make this concrete by revisiting the conceptual model of the industry standard for description of software architectures (IEEE Std 1471-2000 [IEEE-1471, 2000]). The result, the Model-Driven Architectural Views (MDAV) framework, is displayed in Figure 8.1.

8.3.1 MDAV Framework

In Figure 8.1, for the development of a software System, an Architecture is defined that includes the most important design decisions. These are made concrete in an Architectural Description that consists of Models on the one hand, and architectural Views on the other. In the spirit of MDE, models conform to a Metamodel and are used for several (automated) tasks such as, analysis and code generation. Views on the other hand conform to a Viewpoint and are primarily used for communication purposes. Both metamodels and viewpoints are developed to address a certain set of Concerns. A viewpoint prescribes the language to be used to model the architecture. A metamodel specifies the abstract syntax of this language. A view includes diagrams in its primary presentation that represent the associated architectural models.

To allow the use of custom defined DSMLs without the need to specifically develop corresponding graphical representations and editors, we use UML Diagrams. To this end, we map DSML Models to UML Models that are visualised as UML diagrams for inclusion in view documentation with standard UML tooling. Thus, in MDAV the connection between views and models is made through (UML) diagrams. Thanks to this connection, views can be (partly) generated from the same models as the source code; they become model driven.

8.3.2 MDAV Process

In summary, compared to the conceptual model as described by IEEE Std 1471-2000, we add the concept of a diagram that allows to relate a view to a model. Furthermore, in-line with MDE, we explicitly added a metamodel as a description of the modelling language used in a view. Application of the corresponding approach involves three steps: definition of 1) a suitable metamodel, 2) means to create corresponding models, and 3) a mapping to UML.

A suitable metamodel for a particular viewpoint can be defined from scratch or based on an existing ADL that addresses the relevant concern. In the former case, we use a description of the viewpoint (e.g., from Clements et al. [2002a]) and create corresponding elements and relations in the metamodel. In the latter case, we base the metamodel on the ADL's grammar (or other language specification mechanism). Given the typically modest size and simple syntax of ADLs and using appropriate tooling, corresponding metamodels are easily created.

A means to create models associated with the defined metamodel is also required. Depending on the complexity of the associated metamodel different alternatives are suitable, of which we give examples in Section 8.4.

We specify and implement the mapping between the prescribed metamodel and UML using a model transformation language. For several ADLs mappings to UML already exist, that we can specify as model transformations. This allows us to automatically transform an architectural (ADL) model to a UML Model. As such, ADL Models and UML Diagrams can evolve simultaneously.

Although the corresponding UML diagram might not exactly represent the architectural model (e.g., because the latter uses concepts that do not correspond to any UML concept), it is typically complete enough for many communication purposes. This can be concluded by considering the widespread use of UML for architectural documentation in industrial practice (e.g., see Chapter 3 and Lange et al. [2006]). Moreover, in the case of a semantic mismatch, we use stereotypes to indicate the type of ADL element a specific UML element represents.

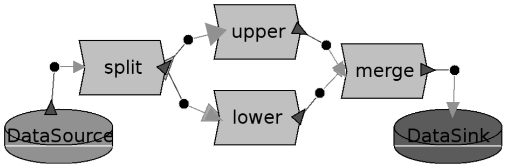


Figure 8.2: C&C model of CaPiTaLiZe (ACME)

8.4 Using MDAV to Generate Views

We applied MDAV to two architectural viewpoints: we defined an appropriate metamodel (i.e., a modelling 'language'), means to create and manipulate associated models, and a mapping to UML,

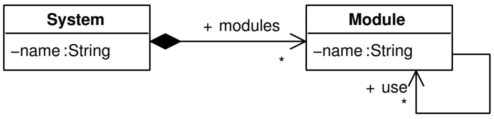
We use the CaPiTaLiZe system, often used in software architecture literature [Allen and Garlan, 1997], as a running example. CaPiTaLiZe transforms a character stream by capitalising alternate characters. A C&C model of CaPiTaLiZe defined using an ADL (ACME [Garlan et al., 2000]) is visualised in Figure 8.2. CaPiTaLiZe is designed as a pipe-and-filter system, with separate components for splitting a stream of characters in two streams (split), (un)capitalising characters (upper, lower), and merging two streams of characters (merge).

The diagram of CaPiTaLiZe's module view is depicted in Figure 8.3(c) . In this UML class diagram we represent architectural modules with UML Packages and use-relations with UML Dependencies, as suggested by Clements et al. [2002a].

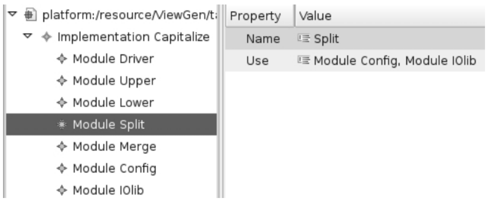
8.4.1 Module-Uses View

Metamodel Module-uses views are based on a special type of dependency relation: the uses relation. As such, these views only contain one type of element and one type of relation [Clements et al., 2002a].

Although UML is well-suited and therefore also typically used in the primary presentation of module views, we developed a small custom metamodel to illustrate MDAV. This MADL metamodel is specified in Figure 8.3(a) using the MOF. In addition to a Module element and use relation it defines an Implementation to consist of a set of modules that may use other modules. Note that this metamodel is different than the MADL metamodel in Figure 6.6(b) on page 113. Although both are used to address the same concerns, their purpose is different (documentation vs. conformance checking). For that reason, the use-relation is modelled by the latter as a first-class modelling element (to allow specifying its conformance).



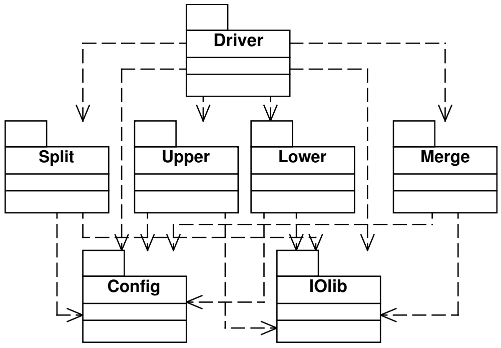
(a) MADL metamodel



platform:/resource/ViewGen/t
Implementation Capitalize
Module Driver
Module Upper
Module Lower
Module Split
Module Merge
Module Config
Module IOlib

Property	Value
Name	Split
Use	Module Config, Module IOlib

(b) MADL model of CaPiTaLiZe in EMF editor



(c) MADL diagram (UML)

Figure 8.3: MADL

```
rule Package {
 from m:MADL!Module
 to p:UML!Package (
   name <- m.name,
   clientDependency <- ds),
 ds: distinct UML!Dependency foreach (um in m.use)(
   client <- m,
   supplier <- um)
}
```

Listing 8.1: Mapping MADL Modules to UML Packages (ATL)

Model creation Using MOF, in principle, only the abstract syntax is defined. Although XMI offers an off-the-shelf mapping to XML, it is not intended to be used directly by software developers [Grose et al., 2002].

For simple metamodels, such as our MADL, we propose to use the editor generated by EMF for the creation and inspection of models. Figure 8.3(b) on the preceding page shows a screenshot of this editor while editing the MADL model for the CaPiTaLiZe system. The top part shows the modules that are defined for this system, while the Properties pane is used to inspect the properties of those modules. This screenshot shows, for instance, that Module Split uses Module Config and Module IOlib.

UML mapping The mapping to UML is based on one of the mappings suggested in [Clements et al., 2002a]. We map Modules to UML Packages and the uses relation to UML Dependencies. We specified this mapping using ATL. A fragment is depicted in Listing 8.1.

In an ATL transformation rule a `from` clause specifies a pattern that is matched by elements of the source model. For each match the target patterns in the `to` clause are instantiated in the target model. In this case, the `Package` rule creates a Package (`p`) and a set of Dependencies (`ds`) for each Module (`m`) in the source model. Using the `distinct ... foreach` construct a Dependency is created for every Module that is used by the Module that matched the rule (`m.use`). For both target elements a set of bindings is specified to initialise their features. The `clientDependency` feature of the created Package (`p`), for instance, is initialised with the set of Dependencies created by this rule as well (`ds`).

The result of applying this transformation to the MADL model of the CaPiTaLiZe system (Figure 8.3(b) on the preceding page), is visualised using a UML tool (Figure 8.3(c) on the previous page).

8.4.2 Component-and-Connector View

Metamodel For C&C views, we define a metamodel for a simple ADL similar to ACME [Garlan et al., 2000], an ADL interchange language that covers the most constructs in a wide range of ADLs.

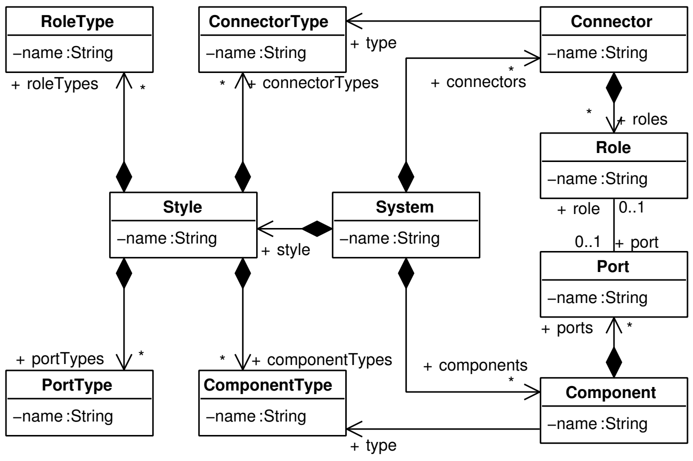
Consider the metamodel for the ADL (CCADL) in Figure 8.4(a) on the following page. Using CCADL the architecture of a System consists of a Style, a set of Components, and a set of Connectors. A component owns a set of Ports via which it interacts with its environment. Similarly a connector owns a set of Roles that define what behaviour is expected from the participants in the interaction the connector represents. By attaching conforming roles and ports, configurations of components and connectors can be created. Finally, the style defines the types of components (ComponentType), connectors (ConnectorType), roles (RoleType), and ports (PortType). The main difference with the CPADL metamodel depicted in Figure 6.6(a) on page 113 is that component, connector, role, and port types are defined on the model level instead of on the metamodel level.

Model creation Again, a straightforward approach to create CCADL models is to use the editor generated by EMF. Figure 8.4(b) on the following page displays a screenshot of this editor, while editing the CaPiTaLiZe CCADL model. When considering the complexity of the CCADL metamodel (compared to the MADL metamodel), it becomes clear that editing models using this editor is inconvenient. Using this editor it is not possible, for instance, to immediately determine the component and connector types and understand their configuration.

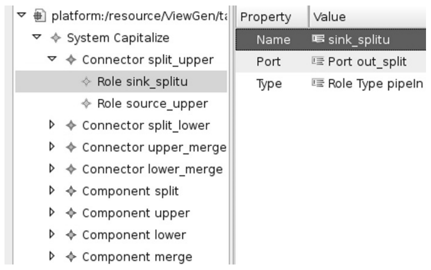
As an alternative, we propose to use a simple XML Document Type Definition (DTD) or schema that allows to describe associated models as simple as possible. A fragment of an XML document conforming to such a DTD describing the same CaPiTaLiZe system is depicted in Listing 8.2 on page 179. Note that the DTD we defined allows to separately specify the configuration of components and connectors as a set of attachments.

If we use simple XML documents to specify systems in CCADL, we separately need to populate a *model* conforming to the CCADL metamodel. Several approaches can be used to populate a model.

One possibility is to develop a so-called injector, a program that parses a file and uses the application programming interface (API) generated by EMF to instantiate a corresponding model. In general, an injector is used to bridge two different domains, in this case the XML and modelware (MOF) domains. In the context of MDE such domains are also referred to as Technological Spaces [Kurtev et al., 2002].



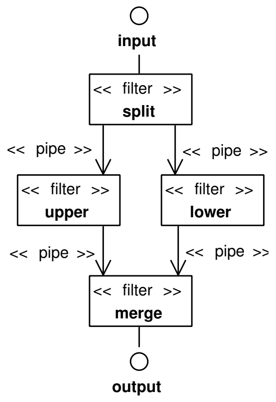
(a) CCADL metamodel



▽ platform:/resource/ViewGen/t	
▽ System Capitalize	
▽ Connector split_upper	
Role sink_splitu	
Role source_upper	
▷ Connector split_lower	
▷ Connector upper_merge	
▷ Connector lower_merge	
▷ Component split	
▷ Component upper	
▷ Component lower	
▷ Component merge	

Property	Value
Name	sink_splitu
Port	Port out_split
Type	Role Type pipeIn

(b) CCADL model of CaPiTaLiZe in EMF editor



(c) CCADL diagram (UML)

Figure 8.4: CCADL

```
...
<System name="Capitalize">
  <Style name="pf">
    <ComponentType name="Filter"/>
    <PortType name="filterOut"/>
    ...
  </Style>
  <Component name="split" type="Filter">
    <Port name="in_split" type="filterIn"/>
    <Port name="out_split" type="filterOut"/>
  </Component>
  ...
  <Connector name="split_upper" type="Pipe">
    <Role name="sink_splitu" type="pipeIn"/>
    <Role name="source_upper" type="pipeOut"/>
  </Connector>
  ...
  <Configuration>
    <Attach port="out_split" role="sink_splitu"/>
    ...
  </Configuration>
</System>
```

Listing 8.2: C&C model of CaPiTaLiZe (XML)

As an alternative, we reuse the XML injector, and XML metamodel (see Figure 8.5 on the following page) provided by the ATL project[1]. Based on an XML document this injector instantiates a model that conforms to the XML metamodel. Subsequently, we transform this model into a model that conforms to the CCADL metamodel using ATL model transformations. The latter approach requires the smallest effort because it reuses existing injectors and metamodels, and only requires us to specify a transformation that maps the XML metamodel to our CCADL metamodel.

The transformation to instantiate a CCADL model based on an (injected) XML source model is straightforward. Listing 8.3 on the next page contains a fragment of this transformation. The rule matches all XML elements named `'Component'`. For each it creates a Component in the CCADL model. The `type` and `name` features are initialised using two helpers, `getType` and `getName`. They navigate the XML model to extract the desired information. For the `type` feature this is another XML Element that, in turn, matches a rule that creates ComponentTypes. The other elements of the CCADL metamodel are instantiated by similar rules.

[1]http://www.eclipse.org/m2m/atl (June 2007)

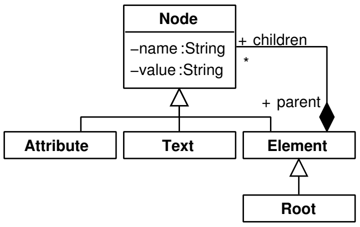


Figure 8.5: XML metamodel

```
rule Component {
  from el:XML!Element(
    el.name='Component')
  to c:CCADL!Component(
    type <- el.getType,
    ports <- el.children->select(e|e.name='Port'),
    name <- el.getName)
}
```

Listing 8.3: Mapping XML Elements to CCADL Components (ATL)

UML mapping The UML representation of components and connectors is based on the strategies for modelling software architecture with UML described by Garlan et al. [2002]. In Figure 8.4(c) on page 178 component and connector types are depicted as stereotypes, components as classes, and connectors as associations. Of the roles and ports we only explicitly show ports that are not connected to a role. These are represented by the input and output interfaces (depending on the type of port), depicted here as small circles connected to the representation of their containing component.

Listing 8.4 shows two rules of the corresponding ATL model transformation. The `Association` rule instantiates an Association (`asoc`) for each Connector (`conn`) in the source model. As our UML tool did not support stereotypes on Associations, we initialise the `name` feature to mimic one. Although not very elegant, this is acceptable when considering the goal of our transformation: generation of diagrams for documentation. In UML an Association has a connection feature that is a set of AssociationEnds. In our case, these represent the Roles of a Connector. For simplicity we assumed a Connector has exactly two Roles. The `connection` feature is initialised to the result of the rule that matches the roles of the Connector. Roles are matched by the `AssociationEnd` rule that creates an AssociationEnd (`aend`) for every matched Role (`r`). The `isNaviagable` feature is initialised depending on whether the matched Role is of type `pipeOut` (true) or not (false). As

```
rule Association {
  from conn:CCADL!Connector
  to asoc:UML!Association(
    name <- '<<' + conn.type.name + '>>',
    connection <-conn.roles)
}
rule AssociationEnd {
  from r:CCADL!Role
  to aend:UML!AssociationEnd(
    isNavigable <- r.type.name='pipeOut',
    participant <-
CCADL!Component->allInstances()->select(c|c.ports->includes(r.port))
 )
}
```

Listing 8.4: Mapping CCADL Connectors to UML Associations (ATL)

such, we control the direction of the Association for representation of the Connector.

Depending on the exact concerns the associated viewpoint addresses, alternative mappings to UML can be implemented similarly, such as a mapping that explicitly shows ports and roles.

8.5 Industrial Application

In this section we discuss a case study in which we applied MDAV to an architectural view in use for a class of control systems. Before discussing the three steps of our approach, we briefly introduce the case study.

ASML, a large manufacturer of equipment for the semi-conductor industry, studies the migration to a new architecture for supervisory machine control (SMC) components. In an advanced manufacturing machine, such as the wafer-scanners developed by ASML, an SMC component is responsible for the coordination of manufacturing activities in order to perform manufacturing requests. In a layered control architecture, an SMC component receives manufacturing requests from components in a higher layer, and coordinates the execution of manufacturing activities by components in a lower layer. Traditionally, the design for SMC systems is based on state transition models. The new approach [Van den Nieuwelaar, 2004] is based on task-resource models.

Metamodel Using the task-resource approach, SMC systems consist for a large part of generic, reusable components defined by a product-line architecture. The remaining application-specific components are generated based on a model of an SMC system in terms of tasks and resources. The associated metamodel is shown in Figure 8.6(a) on page 184.

Task-resource models consist of a static and a dynamic part. The static part models the controlled System by specifying the Behaviours (manufacturing activities) it can perform, the Capabilities this requires, and the Resources (subsystems) it controls to offer those capabilities. The dynamic part models the manufacturing requests the component can perform in terms of (simple, conditional, or compound) Tasks that are of a specific Behaviour. Precedence relations between tasks are used to specify restrictions on execution order.

Based on the metamodel, tools can be developed for the generation of source code, model validation, and other software engineering tasks that can be automated. We used it, for instance, as the target of a model transformation that automates the migration of SMC components from a state-based to a task-resource-based architecture (see Chapter 7).

Model creation In this case, task-resource models were obtained by the automatic migration of legacy SMC models (based on state machines) to models based on the task-resource architecture. As such, a means to create task-resource models directly was not yet required. When SMC systems are developed based on the task-resource approach from scratch, such means would be required. In that case, one of the alternatives presented in the previous section can be used.

An example of a generated task-resource model (as result of the automated migration) inspected using the EMF editor is depicted in Figure 8.6(b) on page 184. This editor was generated based on the metamodel we defined (Figure 8.6(a) on page 184). Apart from this editor there was no (more advanced) editor available for these models.

UML mapping For the documentation of SMC systems based on the task-resource architecture we defined a viewpoint. Due to the lack of a convenient editor to visualise task-resource models and to take advantage of available tooling and experience, the viewpoint prescribes that such models are depicted using UML diagrams. As such, the alignment of the task-resource view documentation with the task-resource models from which code can be generated, involved a mapping of the corresponding metamodel to UML.

For the documentation of a conforming view, separate diagrams are used for the static part and for each of the possible requests of the dynamic part. For the former, a UML Class Diagram is used in which a Class with appropriate Stereotype is used to represent a Behaviour, Capability, or Resource. For the latter, UML Activity Diagrams (one for each request)

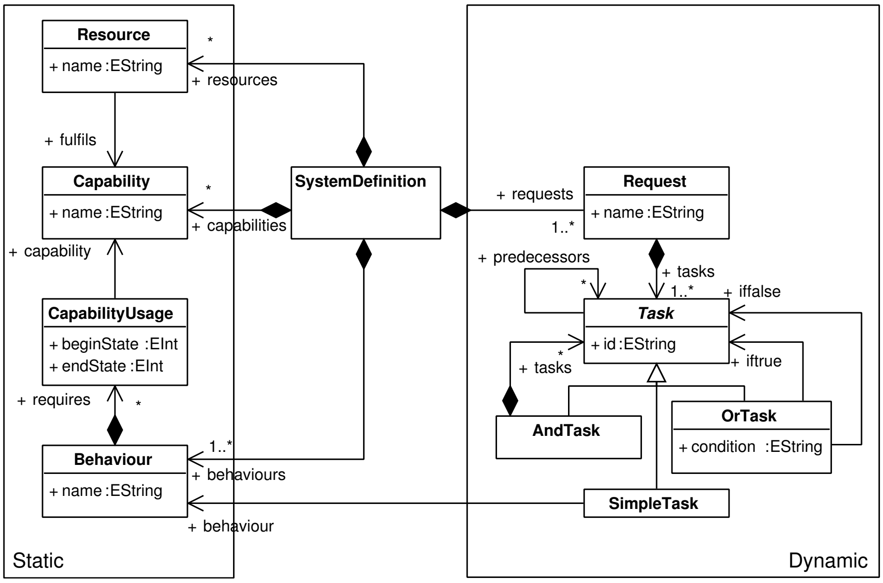
```
rule ActionState {
  from st:TRS!SimpleTask
  to state:UML!ActionState (
    name <- st.behaviour.name,
    stereotype <- stype),
  stype: distinct UML!Stereotype foreach (s in st.sTypes)(
    name <- s,
    baseClass <- 'ActionState'),
  ...
}
```

Listing 8.5: Mapping TRS SimpleTasks to UML ActionStates (ATL)

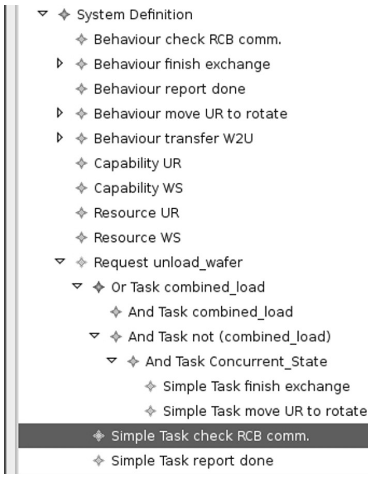
are used that effectively represent tasks and their precedence relations as task graphs. We used ATL to define corresponding mappings from the task-resource metamodel to UML class models for the static part, and to UML activity graphs for the dynamic part. A UML tool visualises these models as a UML Class Diagram, and a UML Activity Diagram, respectively.

As an example, the rule in Listing 8.5 maps a SimpleTask to the element that represented an activity in a UML Activity Diagram: ActionState. The `name` feature of the generated ActionState (`state`) is initialised using the name of the behaviour associated with the SimpleTask (`st`) that matched the rule. The rule also creates a set of Stereotypes (`stype`). In that target element the `sTypes` helper determines the resources required by the behaviour associated with the matched SimpleTask. This set is used to generate a set of ActionState Stereotypes used to initialise the stereotype feature of the generated ActionState.

Application of the transformation we defined to the model partly depicted in Figure 8.6(b) on the following page, results in a class model and an activity graph for each request. One of those is visualised as an Activity Diagram in Figure 8.6(c) on the next page. Tasks are represented by Activities, required resources by Stereotypes on Activities, precedence relations between Tasks by the order of the Activities, fork bars were used to indicate tasks that can be executed concurrently (i.e., tasks without precedence relations), and conditional Tasks (OrTasks) were mapped to choice nodes. As such, to complete the migration, models as the one depicted in Figure 8.6(b) on the following page are used for model-based generation of source code, while diagrams as the one in Figure 8.6(c) on the next page are used for view-based *documentation*. Using model transformations the diagrams for this documentation are generated automatically.

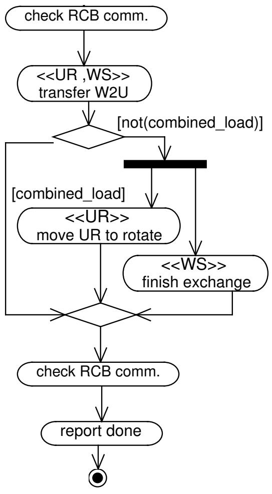


(a) Task-resource metamodel



- System Definition
  - Behaviour check RCB comm.
  - Behaviour finish exchange
  - Behaviour report done
  - Behaviour move UR to rotate
  - Behaviour transfer W2U
  - Capability UR
  - Capability WS
  - Resource UR
  - Resource WS
  - Request unload_wafer
    - Or Task combined_load
      - And Task combined_load
      - And Task not (combined_load)
        - And Task Concurrent_State
          - Simple Task finish exchange
          - Simple Task move UR to rotate
    - Simple Task check RCB comm.
    - Simple Task report done

(b) Task-resource model



(c) Task-resource diagram (UML)

Figure 8.6: Task-resource metamodel, model, and UML representation

8.6 Discussion

Our approach has several benefits. It reduces the effort required for the introduction of MDE approaches by circumventing the need to specifically develop graphical editors for the visualisation of DSML models. Furthermore it allows to introduce an MDE approach gradually; UML diagrams can continue to be used for documentation purposes. As such, in the case of software architecture, it facilitates the integration of ADLs and supporting tools in industrial development processes.

As presented here the approach uses MDA technology for model transformations and metamodelling. The underlying ideas are applicable to other MDE approaches as well: either by using the available transformation and metamodelling technologies for that MDE approach, or by implementing a bridge to MDA. We gave an example of the latter in Section 8.4.2 for XML.

Of course, the diagrams that are generated automatically using our approach, only constitute a minor part of the complete documentation. Architectural views, for instance, typically also document (some of) the rationale and trade-offs that underlie design decisions [Clements et al., 2002a]. In fact, an architectural view can be seen as ‘diagrams + explaining text’. Although the ‘explaining text’ is not automatically updated using our approach, it does provide a starting point for doing so (i.e., the newly generated diagram).

Whether a mapping to UML is feasible, depends on the type of models involved and the documentation requirements. A potential risk of our use of UML, is that the UML semantics might not match with the semantics of the represented (DSML) model elements, resulting in ambiguities. In these cases appropriate stereotypes should be introduced. As an example, consider the stereotypes in Figure 8.4(c) on page 178. These stereotypes are included in the ATL mappings we defined.

In the case that the semantic gap between the involved metamodel and UML is too large to be solved with stereotypes, instead of UML, more generic graph languages such as *dot*[1] and GXL[2] could be used as target of the mapping.

The effort required for specification of the mappings to UML is mainly determined by the complexity and size of the DSML metamodel. Typically, these are relatively small (e.g., compared to UML). Furthermore, such mappings can be either specifically developed (as in the case of task-resource models) or reused (as in the case of ADLs). In the latter case they only need to be specified as a model transformation.

[1]*dot* - Language used by Graphviz (Graph Visualisation Software), see http://www.graphviz.org (June 2007)
[2]GXL - Graph eXchange Language, see http://www.gupro.de/GXL (June 2007)

Our approach focuses on *visualisation* of DSML models. It does not offer visual *editing* for models conforming to complex metamodels. When that is required, editors have to be developed specifically. Technology to partly generate such editors is provided in the Eclipse Graphical Editing Framework[1] (GMF) using EMF. Based on the specification of a concrete syntax and the abstract syntax specified by a metamodel this plugin can generate an editor. However, in the case that only visualisation is required, our approach offers a lightweight alternative.

Another alternative is to simply manually create documentation instead of automatically as in our approach. In that case the diagrams corresponding to some software models are created (drawn) manually using modelling or more generic tools. Obviously, consistency becomes an issue with such an approach.

8.7 Related Work

Fondement and Baar [2005] present an approach to specify (graphical) concrete syntax by extending metamodels. Based on this approach tools could be developed to (partly) generate corresponding editors. Instead, we take advantage of existing UML tools.

Medvidovic et al. [2002] investigate the use of UML in the domain of software architecture. More in particular, they investigate how modelling constructs used in ADLs (i.e., a type of DSML) can be represented using UML. They consider two approaches for using UML to model software architectures: (1) use UML 'as-is', and (2) use UML's extension mechanisms. They conclude that UML has a number of limitations when used to model software architectures. The lack of architectural modelling constructs makes it necessary to adopt specific interpretations of UML model elements or to rely on OCL to constrain the use of model constructs. In this chapter we investigated a third strategy that is based on the definition of metamodels for ADLs and their mapping to UML using model transformations.

Five strategies for representing architectural structure in UML are described by Garlan et al. [2002]. They conclude that there is no single best way to do this. Furthermore, they identify a trade-off between completeness and legibility: strategies that assign different UML model elements for each ADL construct (completeness) tend to be very verbose and hence poorly readable (legibility). One of their recommendations to solve this, is to continue to use ADLs but to provide mappings to object-oriented notations. In the current chapter we specified such mappings using model transformations, which makes them automated.

[1]http://www.eclipse.org/gmf (June 2007)

Where we propose to use MOF for the definition of DSMLs, Dashofy et al. [2005] use XML for the definition of ADLs. It provides generic high-level XML schemas that can be extended for development of ADLs. They leverage the available tool support for XML. As we use MOF, we leverage available UML and MOF tools as well. This enables, for instance, the specification of transformations on a higher level of abstraction by a model transformation language.

8.8 Concluding Remarks

In this chapter we proposed to combine DSML models and UML diagrams for model-driven software documentation. Where MDE approaches typically aim to use DSML models to automatically create source code, our approach complements MDE with the (partial) creation of documentation.

The main motivation for our approach is the observation that although DSMLs have clear advantages over general-purpose modelling languages, it requires considerable effort to develop graphical editors and representations. In particular, the definition and implementation of their concrete syntax or notation is much more involved than that of their abstract syntax, which is supported by technologies, such as MOF and EMF. This is a problem, as graphical representations of models are an essential part of software documentation.

Our approach uses model transformations to (automatically) map DSML models to UML models. These UML models are easily visualised as UML diagrams using available modelling tools. While the DSML models can be used for code generation and other automated software engineering tasks, these diagrams are used in the documentation. As such, our approach allows to optimise both completeness (by the ADL model) and legibility (by the UML diagram) of architecture descriptions. Furthermore, part of the documentation can be automatically updated as the software system evolves.

Application of our approach requires the definition of a DSML metamodel using MOF and mappings to UML using model transformations. This needs to be done once for each DSML used. Furthermore, a means to create associated models is required. We gave several examples for this. Compared to the development of a complete graphical editor for the defined metamodel, our approach is more lightweight.

We evaluated our approach in the domain of software architecture, for which we defined MDAV. It refines the industry standard for architecture documentation (IEEE Std 1471-2000) by linking architectural views (documentation) to architectural models using model transformations and UML. MDAV is easily generalised to other domains. As an example, we discussed an industrial application in the domain of control systems.

Currently we are investigating how the proposed model transformations can best be integrated with existing tooling and development processes. Another problem we are investigating is the (automatic) derivation of metamodels (e.g., based on MOF) from grammars (e.g., based on EBNF). A solution to this problem increases the effectiveness of our approach when applied to existing DSMLs that are not based on MDA technology.

Chapter 9 Conclusion

The goal of this thesis is to investigate techniques that reduce the risks and costs involved in the evolution of software architectures. To structure this problem we introduced four software evolution tasks to be investigated further: evaluation, conformance checking, migration, and documentation.

As a conclusion, in this chapter we revisit the research questions we raised in the introduction of this thesis using our experiences and observations as discussed in the previous chapters. The main question was:

RQ0 *How can the evolution of software architectures be supported?*

Below we address this question by first discussing the subquestions we raised:

RQ1 *How to integrate the support for software evolution tasks in practice, considering the informal use of modelling languages and preference for proven technologies in industry?*

RQ2 *What is the impact of the use of software product lines and platforms on the support for software evolution tasks?*

RQ3 *To what extent can the support for software evolution tasks be automated by the use of model-driven engineering?*

Where the chapters of this thesis are mainly set up according to the software evolution tasks introduced in Chapter 1, the outline of this conclusion is based on our research questions. After a list of our contributions, the results for each of them are discussed in a separate section below. We conclude with a final list of recommendations and future work.

9.1 Contributions

In the process of finding answers to our research questions, we surveyed the current state of the practice of software engineering in industry and developed solutions that support the software evolution tasks we defined. Together with the answers to our research questions, these are the main contributions of this thesis:

- an overview of the software engineering technologies used in industry for the development of embedded software (see Chapter 3);
- an approach for the evaluation of product-line architectures (see Chapter 4);
- a model-driven approach for checking the conformance between state-based and interaction-based behavioural models (see Chapter 5);
- a model-driven and view-based approach for automatically checking the conformance between implementation and architecture (see Chapter 6);
- a model-driven approach for the migration of supervisory machine control architectures (see Chapter 7); and
- a model-driven approach for simultaneous evolution of models and documentation based on views, the Unified Modeling Language[1] (UML), and the Model Driven Architecture[2] (MDA) (see Chapter 8)

One of the distinguishing characteristics of our work lies in the fact that we take into account several advances in software development practices, that is, software product lines and model-driven engineering (MDE). At the same time we consider the impact and application of these approaches in terms of software evolution, which takes up most (up to 90%) of the time, effort, and money of software development projects and organisations [Lientz et al., 1978; Pigoski, 1996]. Furthermore, we explicitly ensured that the methods and techniques we proposed are amenable to be integrated in industrial development practices.

9.2 Integration in Practice (*RQ1*)

An important observation from the survey we reported on in Chapter 3 is the gap between software engineering technologies actually used in industry and those developed by the research community. To reduce this gap,

[1]http://www.uml.org (June 2007)
[2]http://www.omg.org/mda (June 2007)

we continuously considered industrial integration as an important aspect of the solutions we proposed. In particular we aimed at reducing the organisational impact of our solutions and addressed the adoption of MDA standards.

Reducing Organisational Impact Instead of defining new languages and methods, we used existing industrial standards as much as possible (i.e, those related to the MDA) and took into account current industrial practices, such as the informal use of modelling. In general, we aimed at using and extending (similar) technologies as already used in practice. Additionally, we tried to minimise the resources required to apply our solutions.

We did not advocate the use of new languages if not strictly necessary. In the cases where we did define new languages, these are used alongside (Chapter 6) or are mapped to (Chapters 7 and 8) languages already used (i.e., UML). Moreover, for their definition we used the MetaObject Facility[1] (MOF), the metamodelling language of MDA. The advantage is that MOF uses well-known object-oriented concepts to define modelling languages. Furthermore, MOF is supported by an increasing number of tools and open-source implementations are available (e.g., the Eclipse Modeling Framework[2] (EMF)).

Although UML is a well-defined language (at least syntactically), even in organisations where UML is used, models are often very informal. Such models, are more used as illustrative diagrams than precise software specifications. To account for the informal use of modelling languages in general, and in particular that of UML, our solutions for conformance checking (Chapter 5) and migration (Chapter 7) involve a specific normalisation step. Such a step is necessary because our aim to automate these tasks by means of model transformations, requires that input models strictly conform to a metamodel.

Currently, the normalisation step is essential for the application of our model-driven solutions for the software evolution tasks in industry. The main reason is that at present modelling in industry can be characterised as immature [Kleppe et al., 2003], which we also observed during our survey (see Chapter 3). However, when the use of MDE technologies in general, and the associated standards in particular becomes more wide-spread, we expect that the modelling maturity level in industry will rise. As models become more precise, the need for normalisation is reduced.

To increase the potential for integration in practice it is important that a software engineering technique does not require a software development organisation to change much of its current way of working and

[1]http://www.omg.org/mof (June 2007)
[2]http://www.eclipse.org/emf (June 2007)

that the organisational impact in terms of resources of the technique is minimal. Therefore, in Chapter 4 we reduced the number of stakeholders involved in the evaluation approach as much as possible. Furthermore, by reducing the involvement per stakeholder, the organisational impact is minimised. The resulting architecture evaluation process is named Distributed SAAM (DSAAM) and is based on the Software Architecture Analysis Method (SAAM), which is extensively documented [Kazman et al., 1994, 1996; Clements et al., 2002b]). With stakeholder involvement reduced, DSAAM still produced valuable results.

Adoption of MDA Standards Our proposal to generate UML-based documentation from domain-specific language (DSL) models in Chapter 8 enables the adoption of MDE approaches. It is expected that in the future MDE approaches will be more based on domain-specific modelling languages (DSMLs) than on UML [Booch et al., 2004; Bézivin et al., 2005]. This requires a considerable shift from the current wide-spread use of UML. Additionally, the implementation of tool support for DSMLs requires significant effort, for instance, to implement model editors and visualisations. This becomes even more problematic when an MDE approach requires multiple DSMLs to completely specify and subsequently generate applications. In such cases, a mapping to UML can be used, while moving to complete support for those DSMLs with respect to modelling and visualisation tools.

The use of MDA technology, most notably of UML, MOF and XML Metadata Interchange[1] (XMI), throughout the chapters of this thesis further supports the integration of our solutions in practice. The use of these standards takes advantage of existing tooling and skills of present-day software practitioners. However, although their use is an improvement, the level of interoperability as promised by these standards is not achieved. Without investigating the underlying reasons, this was also observed by Lundell et al. [2006]. Among the reasons we encountered during our research are incorrect and incomplete implementation of standards by tool vendors (especially of UML), and the use of different versions of UML, MOF, XMI, and combinations thereof. The consequence is that additional transformations are required on model serialisations before they can be exchanged between different tools. Because of the low-level modifications required in such cases Extensible Markup Language[2] (XML) processing tools, such as Extensible Stylesheet Language Transformations[3] (XSLT), can be used for this.

[1]http://www.omg.org/mda/specs.htm#XMI (June 2007)
[2]http://www.w3.org/XML (June 2007)
[3]http://www.w3.org/TR/xslt (June 2007)

9.3 Software Product Lines (*RQ2*)

In Chapter 3 we signalled a trend towards approaches that allow a more structured form of reuse compared to the ad hoc type of reuse that is typical in industry. In this context product lines and MDE are two important developments. We observed that different companies are organising their software development such that their products are developed as part of a software product line.

The impact of a product-line architecture on our software evolution tasks is two-fold:

- The use of software product lines makes the tasks more complicated because the corresponding product-line architectures are more abstract, apply to multiple products, and, hence, involve a larger number of stakeholders; their scope is larger with respect to the products and stakeholders involved.
- On the other hand, the use of an architecture that applies to a whole set of products improves the return on investment for solutions that apply to all these product-line members.

Scope of Software Product Lines In Chapter 4 we discuss how the use of product-line principles makes software architecture evaluations more complex. The findings of such an evaluation are more based on indirect evidence, as scenarios are identified for product-line members and not for the product line as a whole. Furthermore, a product-line architecture has a much wider scope than a single-product architecture, thereby increasing the number of stakeholders. Our approach takes into account both these effects.

We refined the typical classification of scenarios as either direct (i.e., scenarios that do not require changes to the current architecture) or indirect (i.e., scenarios that do require changes to the current architecture) by distinguishing between two types of direct scenarios: concrete scenarios and floating scenarios. The former are explicitly supported by the product-line architecture, while the latter are not explicitly supported, but not prevented by it as well (remember that a software architecture is both permissive and restrictive with respect to the implementations it allows).

The number of scenarios that will be characterised as floating in a product-line architecture evaluation depends on its maturity. By the maturity scale proposed by Bosch [2002], Océ's product-line architecture can be characterised as a platform. This means that commonalities are identified and separated out as a platform, but that the variabilities are not made explicit. This results in a larger number of floating scenarios. However, if the maturity is raised by the identification of variabilities and their explicit specification in a product-line architecture, a larger number of direct scenarios can be classified as concrete. Thus, a more mature product-line allows a more complete evaluation.

A drawback of scenario-based architecture evaluation approaches is their high organisational impact caused by the involvement of the architecture's stakeholders in a joint evaluation session, which can take up to several days. In the case of a product-line architecture this problem is particularly important. Such architectures have a larger scope resulting in an increased number of stakeholders. Therefore, we restricted the number of stakeholders involved in the joint evaluation session and consulted other stakeholders separately. This significantly reduced the organisational impact of the evaluation.

Increased Return on Investment As we have seen in this thesis, the use of MDE approaches requires considerable effort for the definition of metamodels, and transformation and normalisation rules. For evolution tasks that are carried out on a regular basis, this effort might be justified. Evaluation or conformance checking are examples of tasks that are carried out repeatedly at different points in time. A particular migration, however, is typically carried out only once. The improved reliability of an automatic migration based on MDE is possibly not sufficient to motivate the extra effort required.

The use of product lines allows to justify the required effort also in the case of a migration, as this effort is split over each of the product-line members. As such, the increased return on investment for product-line assets, such as architecture designs and implementations of architectural components, also applies to the model transformations and metamodels developed for the automation of particular software engineering tasks. As an example, in the case of the migration discussed in Chapter 7 for which we defined an approach that is domain specific to some extent, we could reuse the concerns we identified, their corresponding patterns, metamodels, and transformation rules.

9.4 Model-Driven Engineering (*RQ3*)

Our goal was to reduce the risks and costs of architecture evolution. To this end, we aimed at automating our solutions using MDE techniques. Automation is made possible by considering the involved models (in architectural views) as models in the MDE sense, that is, specified using a well-defined (at least syntactically) modelling language. With MDE, modelling languages are defined using metamodels. As such, this requires the creation or reuse

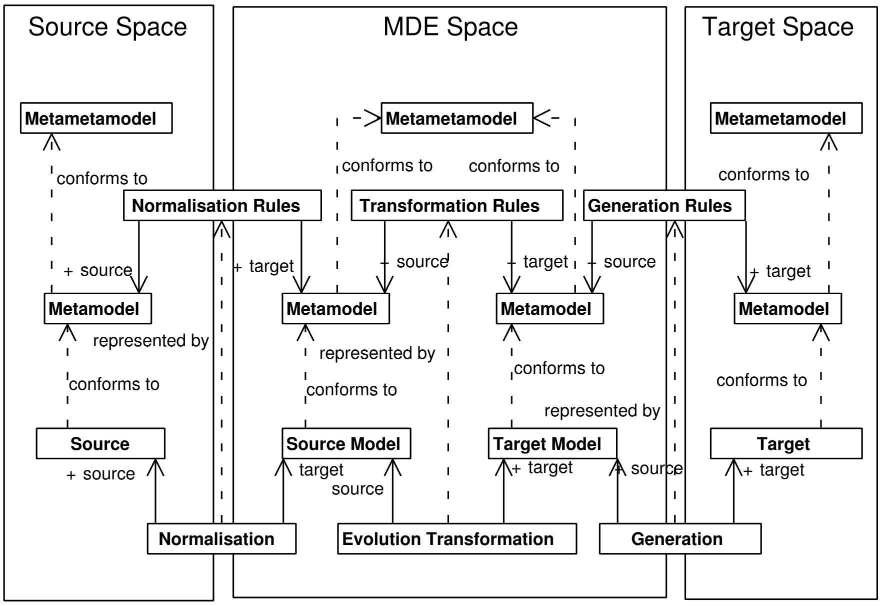


Figure 9.1: Megamodel for model-driven evolution of software architectures

of suitable metamodels. In particular, we employed MDA standards and their supporting tools.

In terms of the effort required for the application of MDE, the automation of a software evolution task involves a trade-off between two aspects: the process of (partly) automating the task, and the subsequent execution of the (partly) automated task. The former determines the costs of following a model-driven approach, while the latter relates to the resulting benefit. We explain all aspects of the deployment of MDE techniques for the software evolution tasks by means of the generic framework in Figure 9.1. In Section 2.3.2 and Figure 2.6 on page 33 we referred to such a framework as a megamodel.

A Megamodel for Model-Driven Evolution of Software Architectures The different MDE solutions for the software evolution tasks we defined and discussed in this thesis lead to the generic megamodel for model-driven evolution of software architectures depicted in Figure 9.1. This megamodel illustrates the artefacts and their relationships involved in the model-driven support of a software evolution task. We revised and extended the two-phased migration process of Figure 7.3 on page 137 such that the processes we applied in the other chapters fit the resulting evolution megamodel as well.

The megamodel involves three technological spaces (see Section 2.3.4): a Source Space, which contains the Source artefacts for a particular evolution task; an MDE Space (i.e., modelware), in which we apply model transformations to support a software evolution task; and a Target Space, in which the Target artefacts are generated. Depending on the evolution context, the Source and Target may also be in the MDE space. Other possibilities include, the XML and grammarware spaces.

The Evolution Transformation is carried out in the MDE Space and transforms a Source Model into a Target Model. It is specified (i.e., represented by) a set of Transformation Rules in the MDE Space. This implies that these rules are defined using a model transformation language. This Evolution Transformation is specific to the task at hand. The Transformation Rules are specified in terms of a source and a target Metamodel associated with the source and target model of the transformation.

As can be seen from this thesis, often no model is available that is suitable to serve as source of the Evolution Transformation. In such cases an additional step is required. Normalisation is the execution of a set of Normalisation Rules with the aim of populating a Source Model in the MDE Space suitable for further transformation using model transformations.

Finally, in the Generation step the Target for the particular evolution task is created according to a set of Generation Rules. The target of the Generation step is in a specific Target Space, for instance, the grammarware or XML space.

All involved model-level artefacts (see Figure 2.3 on page 30), that is, the Source, Source Model, Target Model, and Target conform to a Metamodel. It depends on the type of technological space what kind of Metamodel is used, such as a grammar, MOF metamodel, or XML schema. In turn, these metamodels conform to the governing Metametamodel for that technological space, for instance, MOF in the case of the MDA space.

Normalisation The normalisation step we introduced in several cases is not always fully automated. It is this normalisation step that allows to automate subsequent steps. Normalisation is for a large part context dependent. The amount of normalisation required depends on the type of source artefacts, the modelling maturity level, modelling conventions, the scope of the source language, and the transformation rules.

When the Source Space of the software evolution task is not the same as the MDE Space, normalisation at least includes a bridge between that Source Space and the particular MDE Space (e.g., MDA). In some cases normalisation involves not more than that bridge. If the source is based on a well-defined language the bridge can be fully automated.

Normalisation in Chapter 6 was slightly more involved. Here, after the bridge between the grammarware and MDA spaces, for which we reused existing grammarware to XML and XML to MDA bridges, also some abstraction steps were required. We specified these abstraction steps using model transformations. Hence, this normalisation step was also fully automated.

In other cases normalisation is more difficult and requires replacement of custom annotations (see Chapter 5), the application of a UML profile, or the application of standard idioms for particular concerns (see Chapter 7). These cases involve manual effort for normalisation that requires domain knowledge. As an example, consider the use of UML, a general-purpose language (GPL), in Chapter 7. UML allows to express a particular concern using a multitude of different idioms. Therefore, additional constraints and modelling conventions are required to reduce the complexity of model transformations. Otherwise such transformations have to take into account too many possible idioms for a particular concern, as we illustrated in Chapter 7. As a solution we identified the relevant concerns in the domain of supervisory machine control (SMC) systems. For each concern, we defined a single corresponding design idiom, which conforms strictly to a corresponding metamodel. We defined a UML-SMC profile, which consists of stereotypes and well-formedness rules in the Object Constraint Language[1] (OCL) corresponding to these design idioms. Normalisation involves applying stereotypes to appropriate model elements and modifying the source model in such a way that concerns are consistently addressed by their corresponding design idiom without violating any of the well-formedness rules of the profile.

For complex normalisations, we identified a trade-off between complex transformation rules to account for a large idiom of possible input patterns, or a more extensive normalisation procedure to account for a large number of restrictions on the source models (i.e., to limit the size of the source language).

So, even in cases where the source of an evolution task is a valid UML model (i.e., a model that conforms to the UML metamodel), normalisation can be required to restrict the number of possible source idioms resulting in a less complex evolution transformation. This was necessary in Chapter 7 and to a lesser extent also in Chapter 5.

The need for restricting UML like this, raises the question whether the use of a DSML defined using MOF in such cases wouldn't be more appropriate. This is a matter of ongoing debate between two schools of thought in the MDE community [France and Rumpe, 2007]. One argues in favour of the use of an extensible general-purpose modelling language, while the other promotes the definition of DSMLs. In current practice, however, the lack of

[1] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (June 2007)

sufficient tool support for the definition of DSMLs and the wide-spread use of UML are for companies often sufficient reasons for using UML.

Evolution Transformation Except for evaluation, our solutions for the software evolutions tasks were at least partly automated by specifying them as model transformations in the Atlas Transformation Language [Jouault and Kurtev, 2005] (ATL). In our solution for the conformance checking tasks in Chapter 6, we had two source models. In general, we specified the different steps of an evolution task in separate model transformations. Typically, multiple of such transformations are required.

Now that MDE approaches that involve the definition and application of model transformations get more in use, companies might want to reconsider their use of UML. In our research we experienced that the use of UML results in complicated definitions of model transformations. The reasons for this are the aforementioned size of the UML metamodel, as well as its complexity and the fact that most UML modelling tools only partially implement the UML specification or, even worse, incorrectly. The former relates to the trade-off we identified between the number of source model restrictions involved in the normalisation procedure and the complexity of subsequent model transformations.

Because the current state of modelling in industry is such that languages as UML are only used informally and free-form box-and-line diagrams are also often used, the application of model-driven approaches in industrial contexts typically requires the definition of metamodels. Of course, transformation rules also need to be specified. The extent to which the effort that this requires is justified depends on the particular context. As discussed above in the case of product lines the return on investment for this effort is increased. Here, the use of standards for modelling, metamodelling, and model transformations, offers a very strong benefit. They enable, for instance, the creation of repositories to share these MDE artefacts among different projects, product lines, and companies. In fact, we contributed the model transformations we defined in Chapter 5 for the generation of a state model from a set of scenarios to such a repository[1]. In other cases, we reused metamodels (e.g., metamodels for DOT and XML) and transformations (e.g., DOT to text) available from repositories ourselves as well.

Generation The final (code) generation step is well-studied in the MDE literature. For that reason, we decided not to focus on this step in this thesis, directing our attention to the normalisation and transformation steps instead. Nevertheless, after the execution of the evolution transformation(s),

[1]http://www.eclipse.org/gmt/atl/atlTransformations (June 2007)

typically, some output needs to be generated, possibly in a different technological space. The result can be, for instance, source code, diagrams, or an XML representation of the target model. Examples of such generation steps in this thesis include the following:

- In Chapter 5 generation encompasses serialising the result of the evolution transformation using XMI into XML with the aim of loading the target model in a UML modelling tool for visualisation. Essentially, the XMI standard provides the necessary generation rules in this case, and acts as a bridge between the MDA and XML space.
- In Chapter 6 we generate DOT code in the grammarware space from the target model. The generation rules include rules to map the target model of the evolution transformation to a (MOF-based) DOT model and rules to generate DOT source code from that DOT model. We defined the former ourselves, and reused the latter from a repository of model transformations.
- Finally, in Chapter 8, the goal of the evolution task is to visualise DSML models as UML diagrams to be included in documentation. Here, the generation step is the transformation of a UML model in some graphical format, such as Scalable Vector Graphics[1] (SVG) (in the XML space) or PostScript (in the grammarware space).

9.5 Support for Evolution of Software Architectures (*RQ0*)

Having looked at the three subquestions, we return to the main question, how to support the evolution of software architectures. We first revisit the four software evolution tasks, and then discuss the scope of the industrial case studies we conducted.

Software Evolution Tasks By validating the research results by means of industrial case studies, an evaluation of the applicability of our techniques in to the software evolution tasks in industrial practice was obtained.

For evaluations, in contrast with other approaches (e.g., Gallagher [2000]; Olumofin and Mišlić [2006]), DSAAM specifically takes into account the difference in scope (i.e., with respect to products and stakeholders) between single-product architectures and product-line architectures. In fact, Océ used the results of our evaluation to decide to continue with the development (evolution) of their reference (product-line) architecture.

[1]http://www.w3.org/Graphics/SVG (June 2007)

We discussed two approaches for conformance checking. One is fully automatic, while the other requires a manual comparison. We applied the latter in Chapter 5 to the embedded software for copiers developed by Océ and a small ATM example. In both applications we detected inconsistencies that would have been difficult to detect without our support. Although the actual comparison is manual, it is made possible by our automatic mapping between two types of architectural models, which we specified using model transformations. In Chapter 6 we focused on also automating the actual comparison step.

For the migration task our automatic, model-driven solution offers clear benefits with respect to the alternative, a manual migration. The need for domain experts is reduced and the necessary definition of a suitable meta-model increases the understanding of the migration itself and the target architecture. Finally, because the defined transformation and normalisation rules are generic, they can be reused for the migration of other SMC components. This was illustrated by the migration of a second SMC component, for which we only had to define a few extra transformation rules to include features of the target architecture that were not relevant in the first case. Here, the use of product-line principles for the development of these SMC components, justifies the effort required for applying a model-driven migration approach.

Our solution for the documentation task offers an alternative for developing a complete (graphical) notation and corresponding editor for a DSML. Although it might not always be possible to define a suitable mapping to UML due to the semantic gap between UML and the DSML, our solution has the benefit of being more light-weight. As such, our approach is particularly suited for situations where graphical editing of DSML models is not (yet) required, for example, when a company is gradually migrating from UML to full DSML support.

Our results demonstrate the applicability of model-driven solutions to specific software evolution tasks. For the software evolution tasks we considered, we proposed solutions that take into account product-line architectures (opposed to single-product architectures), aim to reduce organisational impact, or are model-driven. Furthermore, we extend and use technologies that have already proven their applicability in practice, such as SAAM and MOF.

Embedded Software Although our solutions were investigated in the context of concrete (industrial) problems, our evaluations show that they can be applied (to some extent) to our software evolution tasks for a broader class of systems. In the introduction we also raised the question whether our results only apply to the evolution of *embedded* software. To answer this question we have to decide whether a software system's 'embeddedness' is relevant from the perspective of the software evolution tasks we identified.

Our work applies to a special type of software in terms of the case studies we conducted; all were in the domain of embedded software. A common perception is that developing embedded software is different from developing other kinds of software because of some specific characteristics: embedded software has a dedicated function, and is embedded in, and reactive and logically connected to a physical system composed of hardware (e.g., mechanical, electronic, or optical components) and software. These characteristics make that embedded software has some specific properties that make developing embedded software different from a *technical* point of view. As an example, in many cases real-time constraints play an important role, as well as size of memory footprints. Furthermore, embedded software often needs to comply to safety constraints (in the case it controls a physical system that might cause physical damage).

So, indeed embedded software has many specific characteristics. However, most important for software evolution, the topic of our research, is the fact that this type of software is embedded in a physical system. Although we cannot conclude from this that software evolution is different for embedded systems, it does make evolution unavoidable. In fact, the two software evolution laws discussed in Section 2.1 only apply to a special class of systems. Lehman [1980] defines this class of, so-called, *E-type* systems, as the class of systems of which the specification includes a model of the 'real' world. The embedded systems such as those studied in our case studies are prototypical examples of E-type systems. Therefore, we conjecture that our results are valid for E-type systems in general, and not just for embedded systems.

9.6 Future Work and Recommendations

In this thesis we investigated how to support four different software evolution tasks. To this end, we defined solutions that are model-driven and take into account the use of product lines. To enable industry to integrate our solutions in their development processes, we minimised their organisational impact by reusing proven technologies and standards as much as possible and limiting the required additional effort. In most cases we interpreted the software evolution tasks as model transformation problems and provided suitable transformation rules, which can be executed automatically.

In the chapters of this thesis we raised many issues to be investigated further that include, supporting software evolution tasks in other technological spaces, development of hybrid approaches, and management and evolution of modelware artefacts. To conclude we briefly revisit them below.

We primarily used MDA model transformations in ATL to support software evolution tasks. Similar support can also be developed in other technological spaces. In the grammarware space, for instance, also formalisms and tools are available for the definition of languages and transformations, such as the ASF+SDF Meta-Environment [Klint, 1993] and Stratego/XT [Visser, 2004]. As the processes we defined for the support of the different evolution tasks are technology independent, our work provides the starting point for developing similar support by using and combining other technologies. This raises interesting research questions with respect to which technological space is best suited for development of support for a specific evolution task and how to better combine languages and transformations defined in different technological spaces.

Assuming that source code remains in the grammarware and software models remain in the modelware technological spaces, this combination of artefacts from different technological spaces is unavoidable for most solutions for software evolution tasks. Unfortunately, the required bridges currently need to be specifically developed, at least partially. The problem with the bridges we used is that they are defined on the metamodel level. For such bridges to be generic and reusable they should be defined on the metametamodel level. In fact, for MDA to XML such a bridge is already available in the form of XMI. A similar bridge between grammarware and MDA is essential for combining these two technological spaces. This bridge would map EBNF to MOF such that EBNF grammars can be automatically transformed in corresponding MOF metamodels and vice versa, as well as the programs and models conforming to those grammars and metamodels.

Our experience shows that for automatic support of a particular software evolution task multiple model transformations are required. Each model transformation involves its own transformation rules, source and target models, and corresponding metamodels. Moreover, often the support of evolution tasks also involves operations outside the MDA space, such as XSLT transformations in the XML space, or *sed* and *Perl* scripts. This makes that the management of all the involved artefact and transformations steps requires special attention. Although this problem was not the focus of our research, in one case we used a build tool (Ant) to solve this problem. However, with this approach, the required configuration files also tend to get very complex. As such, this problem calls for additional support, which requires additional research.

Means for the management of modelware artefacts are essential for successful application of our solutions in industry. Similar to other software development artefacts this modelware is also expected to evolve. Interesting possibilities to minimise the required evolution of such artefacts by raising their generality, are higher-order transformations. Such transformations that have another transformation as source or target model, can be used for the conformance checking task, for instance, to generate conformance checking transformations from the metamodels associated with the involved models. This requires that there is a metamodel for the transformation language, which is the case for ATL.

Finally, for further investigation of the issues discussed above, industrial case studies should play an essential role as they did in this thesis. The focus on real industrial problems enabled us to discover and investigate difficulties that are inherent to industrial practice. As an example, the need for normalisation, which plays a prominent role in this thesis, could not have been investigated without such case studies.

Bibliography

Abowd, Gregory, Robert Allen, and David Garlan. Using style to understand descriptions of softwar architecture. *ACM SIGSOFT Software Engineering Notes*, 18(5):pages 9–20, 1993.

Al-Ekram, Raihan and Kostas Kontogiannis. An XML-based framework for language neutral program representation and generic analysis. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 42–51. IEEE Computer Society, 2005.

Aldrich, Jonathan, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 187–197. IEEE Computer Society, 2002.

Allen, Robert and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering. (ICSE 1994)*, pages 71–80. IEEE Computer Society, 1994.

Allen, Robert and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):pages 213–249, 1997.

Amyot, Daniel and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):pages 61–94, 2003.

Atkinson, Colin and Thomas Küne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):pages 36–41, 2003.

ATLAS group. *ATL User Manual*. LINA & INRIA, Nantes, 2006. http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf.

Badros, Greg J. Javaml: a markup language for java source code. *Computer Networks*, 33(1–6):pages 159–177, 2000.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

Baxter, Ira D., Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 625–634. IEEE Computer Society, 2004.

Bengtsson, PerOlof, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1–2):pages 129–147, 2004.

Bézivin, Jean. On the unification power of models. *Software and Systems Modelling*, 4(2):pages 171–188, 2005.

Bézivin, Jean. Model driven engineering: An emerging technical space. In *Generative and Transformational Techniques in Software Engineering, International Summer School, (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, 2006.

Bézivin, Jean, Mikäel Barbero, and Frédérique Jouault. On the applicability scope of model driven engineering. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 3–7. IEEE Computer Society, 2007.

Bézivin, Jean and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE Computer Society, 2001.

Bézivin, Jean, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture: European MDA Workshops*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer-Verlag, 2005.

Bontemps, Yves, Patrick Heymans, and Pierre-Yves Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Transactions on Software Engineering*, 31(12):pages 999–1014, 2005.

Booch, Grady, Alan Brown, Sridhar Iyengar, James Rumbaugh, and Bran Selic. An MDA manifesto. In Frankel, David S. and John Parodi, editors, *The MDAJournal: Model Driven Architecture Straight from the Masters*, chapter 11. Meghan-Kiffer Press, 2004.

Bosch, Jan. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.

Bosch, Jan. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the 2nd International Conference on Software Product Lines (SPLC 2)*, volume 2379 of *Lecture Notes in Computer Science*, pages 257–271. Springer-Verlag, 2002.

Bosch, Jan and Peter Molin. Software architecture design: evaluation and transformation. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS'99)*, pages 4–10. IEEE CS, 1999.

Bril, R.J., R.L. Krikhaar, and A. Postma. Architectural support in industry: a reflection using C-POSH. *Journal of software maintenance and evolution: research and practice*, 17:pages 3–25, 2005.

Bröhl, A.P. and W. Dröschel. *Das V-Modell. Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenbourg-Verlag, München, 2nd edition, 1995.

Brooks, Frederick P., Jr. *The Mythical Man-Month*. Addison-Wesley, 1975.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture:a system of patterns*. John Wiley & Sons, 1996.

Buttazzo, G.C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 2002.

Chen, Peter Pin-Shan. The entity-relationship model – toward a unified view of data. *ACM Transactions Database Systems*, 1(1):pages 9–36, 1976.

Clarke, Edmund M., Jr., Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 1999.

Cleaveland, J. Craig. Building application generators. *IEEE Software*, 5(4):pages 25–33, 1988.

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures:Views and Beyond*. Addison-Wesley, 2002a.

Clements, Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. Addison-Wesley, 2002b.

Clements, Paul and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

Cornelissen, Bas, Bas Graaf, and Leon Moonen. Identification of variation points using dynamic analysis. In *Proceedings of the 1st International Workshop on Reengineering towards Product Lines (R2PL 2005)*, pages 9–13. 2005.

Cornelissen, Bas, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 213–222. IEEE Computer Society, 2007.

Czarnecki, K. and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):pages 621–645, 2006.

Czarnecki, Krzysztof and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

Damm, Werner and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:pages 45–80, 2001.

Dashofy, Eric M., André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):pages 199–245, 2005.

Deelstra, Sybren, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):pages 173–194, 2005.

Deelstra, Sybren, Marco Sinnema, Jilles van Gurp, and Jan Bosch. Model driven architecture as approach to manage variability in software product families. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003)*, number TR-CTIT-03-27 in CTIT Technical Report, pages 109–114. University of Twente, 2003.

Dijkstra, Edsger W. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):pages 341–346, 1968.

Dijkstra, Edsger W. On the role of scientific thought. Published in Dijkstra [1982], 1974. EWD447.

Dijkstra, Edsger W. *Selected writings on computing: a personal perspective*. Springer-Verlag, 1982.

Dinther, Y. van, W. Schijfs, F. van den Berk, and K. Rijnierse. Architectural modeling: Introducing the Architecture MetaModel. In *Landelijk Architectuur Congres*. SERC, Utrecht, The Netherlands, 2001.

Dobrica, L. and E. Niemelä. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):pages 638–653, 2002.

Dohmen, L. A. J. and L. J Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES2002)*, volume 2559 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, 2003.

Doyle, Duncan, Hans Geers, Bas Graaf, and Arie van Deursen. Migrating a domain-specific modeling language to MDA technology. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ateM 2006)*, number 1 / 2006 in Mainzer Informatik-Berichte, pages 47–54. Johannes Gutenberg-Universität Mainz, 2006.

D'Souza, Desmond Francis and Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

Eden, A.H., Y. Hirshfeld, and R. Kazman. Abstraction classes in software design. *IEE Proceedings Software*, 153(4):pages 163–182, 2006.

Eden, Amnon H. and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 149–159. IEEE Computer Society, 2003.

Emam, Khaled El, Jean-Normand Drouin, and Walcelio Melo. *The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society, 1997.

Endres, Albert and Dieter Rombach. *A Handbook of Software and Systems Engineering*. Addison Wesley, 2003.

Fahmy, Hoda and Richard C. Holt. Software architecture transformations. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM 2000)*, pages 88–96. IEEE Computer Society, 2000a.

Fahmy, Hoda and Richard C. Holt. Using graph rewriting to specify software architectural transformations. In *Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 187–196. IEEE Computer Society, 2000b.

Favre, Jean-Marie. Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of Thotus the Baboon. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005a.

Favre, Jean-Marie. Foundations of model (driven) (reverse) engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005b.

Fondement, Frédéric and Thomas Baar. Making metamodels aware of concrete syntax. In *Proceedings of the 1st European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2005)*, volume 3748 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2005.

Forward, Andrew and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering (DocEng 2002)*, pages 26–33. ACM Press, 2001.

France, Robert and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FoSE 2007)*, pages 37–54. IEEE Computer Society, 2007.

Gallagher, Brian P. Using the architecture tradeoff analysis method to evaluate a reference architecture: A case study. Technical Report CMU/SEI-2000-TN-007, Carnegie Mellon University, Software Engineering Institute, 2000.

Garlan, David, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 1995)*, pages 179–185. ACM Press, 1995.

Garlan, David, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44:pages 23–49, 2002.

Garlan, David, Robert T. Monroe, and David Wile. ACME: architectural description of component-based systems. In *Foundations of component-based systems*, pages 47–67. Cambridge University Press, 2000.

Garlan, David and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 2, pages 1–39. World Scientific Publishing Company, 1993.

Gerber, Anna, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer-Verlag, 2002.

Gohari, P. and W.M. Wonham. Reduced supervisors for timed discrete-event systems. *IEEE Transactions on Automatic Control*, 48(7):pages 1187–1198, 2003.

Graaf, Bas. Model-driven evolution of software architectures. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 357–360. IEEE Computer Society, 2007.

Graaf, Bas, Marco Lormans, and Hans Toetenel. Software technologies for embedded systems: An industry inventory. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES 2002)*, volume 2559 of *Lecture Notes in Computer Science*, pages 453–465. Springer-Verlag, 2002.

Graaf, Bas, Marco Lormans, and Hans Toetenel. Embedded software engineering: The state of the practice. *IEEE Software*, 20(6):pages 61–69, 2003.

Graaf, Bas and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 115–126. IEEE Computer Society, 2007a.

Graaf, Bas and Arie van Deursen. Using MDE for generic comparison of views. In *Proceedings of the 4th International Workshop on Model Design, Verification and Validation (MoDeVVa 2007)*, pages 57–66. INRIA, 2007b.

Graaf, Bas and Arie van Deursen. Visualisation of domain-specific modelling languages using UML. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2007)*, pages 586–595. IEEE Computer Society, 2007c.

Graaf, Bas, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture – industrial experience report. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 354–363. IEEE Computer Society, 2005.

Graaf, Bas, Sven Weber, and Arie van Deursen. Migrating supervisory control architectures using model transformations. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 151–160. IEEE Computer Society, 2006.

Graaf, Bas, Sven Weber, and Arie van Deursen. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2007. Doi: 10.1016/j.jss.2007.06.007.

Gray, Jeff, Jing Zhang, Suman Roychoudhury, Hui Wu, Rajesh Sudarsan, Aniruddha, Sandeep Neema, Feng Shi, and Ted Bapty. Model-driven program transformation of a large avionics framework. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE 2004)*, pages 361–378. Springer-Verlag, 2004.

Greenfield, Jack, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

Grose, Timothy J., Gary C. Doney, and PhD. Stephan A.Brodsky. *Mastering XMI. Java programming with XMI, XML, and UML*. John Wiley & Sons, 2002.

Han, Minmin, Christine Hofmeister, and Robert L. Nord. Reconstructing software architecture for J2EE web applications. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 67–78. IEEE Computer Society, 2003.

Hatley, Derek J. and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing, 1987.

Hofmeister, C., R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.

Hofmeister, Christine, Philipe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. Generalizing a model of software architecture design from five industrial approaches. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 77–88. IEEE Computer Society, 2005.

Horowitz, Ellis, Alfons Kemper, and Balaji Narasimhan. A survey of application generators. *IEEE Software*, 2(1):pages 40–54, 1985.

Humphrey, Watts S. *Managing the software process*. Addison-Wesley, 1989.

IEEE-1219. IEEE standard for software maintenance. IEEE Std 1219–1998, 1998.

IEEE-1471. IEEE recommended practice for architectural description of software intensive systems. IEEE Std 1471–2000, 2000.

Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

Jacobson, Ivar, Martin Griss, and Patrick Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

Jansen, Anton. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 109–120. IEEE Computer Society, 2005.

Jouault, Frédéric and Ivan Kurtev. Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. 2005.

Kazman, Rick, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):pages 47–55, 1996.

Kazman, Rick, Len Bass, and Mark Klein. The essential components of software architecture design and analysis. *Journal of Systems and Software*, 79(8):pages 1207–1216, 2006.

Kazman, Rick, Len Bass, Mike Webb, and Gregory Abowd. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering. ICSE 1994*, pages 81–90. IEEE Computer Society, 1994.

Kitchenham, Barbara, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):pages 52–62, 1995.

Klein, Mark H., Rick Kazman, Len Bass, Jeromy Carrierea, Mario Barbacci, and Howard Lipson. Attribute-based architecture styles. In *Proceedings of the 1^{st} Working IFIP Conference on Software Architecture (WICSA 2001)*, pages 225–243. 1999.

Kleppe, Anneke, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

Klint, Paul. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering*, 2(2):pages 176–201, 1993.

Klint, Paul, Ralf Lämmel, and Chris Verhoef. Towards an engineering discipline for grammarware. *Transactions on Software Engineering and Methodology*, 14(3):pages 331–380, 2005.

Kobryn, Cris. UML 2001: a standardization odyssey. *Communications of the ACM*, 42(10):pages 29–37, 1999.

Krikhaar, René L. *Software architecture Reconstruction*. Ph.D. thesis, Universiteit van Amsterdam, 1999.

Kruchten, Philipe, Henk Obbink, and Judith Stafford. The past, present, and future of software architecture. *IEEE Software*, 23(2):pages 22–30, 2006.

Kruchten, Philippe B. The 4+1 view model of architecture. *IEEE Software*, 12(6):pages 42–50, 1995.

Kruchten, Phillipe. *The Rational Unified Process*. Addison-Wesley, 1998.

Krueger, Charles W. Software reuse. *ACM Computing Surveys*, 24(2):pages 131–183, 1992.

Kurtev, Ivan, Jean Bézivin, and Mehmet Aksit. Technological spaces: an initial appraisal. In *Confederated International Conferences CoopIS, DOA, and ODBASE 2002, Industrial Track*. Springer-Verlag, 2002.

Kuvaja, Pasi, Jouni Similä, Lech Krzanik, Adriana Bicego, Samuli Saukkonen, and Günter Koch. *Software Process Assessment and Improvement: The BOOTSTRAP Approach*. Blackwell Publishers, 1994.

Lam, Vitus S.W. and Julian Padget. Analyzing equivalences of UML statechart diagrams by structural congruence and open bisimulations. In *Proceedings of the 2003 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2003)*, pages 137–144. IEEE Computer Society, 2003.

Lange, Christian F.J., Michel R.V. Chaudron, and Johan Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):pages 40–46, 2006.

Lehman, M. M. Laws of program evolution - rules and tools for programming management. In *Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail?*, pages 11/1 – 11/25. 1978. Reprinted as Chapter 12 in [Lehman and Belady, 1985].

Lehman, M. M. and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press, 1985.

Lehman, Meir M. Programs, life cycles, and laws of software evolution. *IEEE Proceedings*, 68(9):pages 1060–1076, 1980.

Liang, Hongzhi, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006)*, pages 5–11. ACM, 2006.

Lientz, B. P., E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):pages 466–471, 1978.

Liu, C.L. and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):pages 46–61, 1973.

Lundell, Björn, Brian Lings, Anna Persson, and Anders Mattsson. UML model interchange in heterogeneous tool environments: An analysis of adoptions of XMI 2. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, number 4199 in Lecture Notes in Computer Science, pages 619–630. Springer-Verlag, 2006.

Lutz, Robyn R. and Gerald C. Gannod. Analysis of a software product line architecture: an experience report. *The Journal of Systems and Software*, 66(3):pages 253–267, 2003.

Medvidovic, Nenad, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):pages 2–57, 2002.

Medvidovic, Nenad and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5:*

Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, pages 60–76. Springer-Verlag, 1997.

Mellor, Stephen J., Anthony M. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):pages 14–18, 2003.

Mens, Kim. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. Ph.D. thesis, Vrije Universiteit Brussel, 2000.

Mens, Tom and Pieter van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:pages 125–142, 2006.

Mernik, Marjan, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):pages 316–344, 2005.

Monroe, R.T., A. Kompanek, and D. Melton, R.; Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):pages 43–52, 1997.

Murphy, Gail C., David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM Press, 1995.

Olumofin, Femi G. and Vojislav B. Mišlić. Extending the ATAM architecture evaluation to product line architectures. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 45–56. IEEE Computer Society, 2006.

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification. http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

OMG. OMG Unified Modeling Language Specification, Version 1.4. http://www.omg.org/docs/formal/01-09-67.pdf, 2007a.

OMG. Unified Modeling Language: Superstructure, version 2.1.1. http://www.omg.org/docs/formal/07-02-05.pdf, 2007b.

Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):pages 1053–1058, 1972.

Partsch, H. and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):pages 199–236, 1983.

Perry, Dewayne E. and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):pages 40–52, 1992.

Pigoski, Thomas M. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, 1996.

Potts, Colin. Software engineering research revisited. *IEEE Software*, 10(5):pages 19–28, 1993.

Poulin, J. S. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

Ramadge, P.J. and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):pages 206–230, 1987.

Reveliotis, Spyros A. *Real-Time Management of Resource Allocation Systems. A Discrete Event Systems Approach*, volume 79 of *International Series in Operations Research & Management Science*. Springer-Verlag, 2005.

Sabuncuoglu, I. and M. Bayiz. Analysis of reactive scheduling problems in a job-shop environment. *European Journal of operational research*, 126:pages 567–586, 2000.

Schäfer, Timm, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):pages 357–369, 2001.

Schmidt, Douglas C. Model-driven engineering. *IEEE Computer*, 39(2):pages 25–31, 2006.

Seidewitz, Ed. What models mean. *IEEE Software*, 20(5):pages 26–32, 2003.

Selic, Bran. The pragmatics of model-driven development. *IEEE Software*, 20(5):pages 14–25, 2003.

Selic, Bran, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

Sendall, Shane. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):pages 42–45, 2003.

Shaw, Mary, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstarctions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):pages 314–335, 1995.

Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Software Engineering Institute. Published software architecture definitions. http://www.sei.cmu.edu/architecture/published_definitions.html, 2006.

Soni, D., R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 1995)*. ACM Press, 1995.

Sonnenberg, C. J. *Improving software maintainability: A case study*. Master's thesis, Technische Universiteit Eindhoven, 2005.

Spanjers, Hans, Maarten ter Huurne, Dan Bendas, Bas Graaf, Marco Lormans, and Rini van Solingen. Tool support for distributed software engineering. In *Proceedings of the 1st International Conference on Global Software Engineering (ICGSE 2006)*, pages 187–198. IEEE Computer Society, 2006.

Stevens, Perdita. On associations in the Unified Modelling Language. In *Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools (≪UML≫ 2001)*, volume 2185 of *Lecture Notes in Computer Science*. 2001.

Swanson, E. Burton. The dimensions of maintenance. In *Proceedings 2nd International Conference on Software Engineering (ICSE 1976)*, pages 492–497. IEEE Computer Society, 1976.

Terekhov, Andrey A. and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):pages 111–124, 2000.

PROGRESS. Embedded systems roadmap 2002: Vision on the technology for the future of PROGRESS. Technical report, Technology Foundation (STW), 2002. http://www.stw.nl/Programmas/Progress/ESroadmap.htm.

Van den Brand, M.G.J., A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC*

2001), volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

Van den Nieuwelaar, N.J.M. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Ph.D. thesis, Technische Universiteit Eindhoven, 2004.

Van den Nieuwelaar, N.J.M., J.M. van de Mortel-Fronczak, and J.E. Rooda. Design of supervisory machine control. In Glover, Keith and Jan Maciejowski, editors, *Proceedings of the European Control Conference ECC 2003*. 2003.

Van der Aalst, Wil, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):pages 1128–1142, 2004.

Van Deursen, A., C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 4)*, pages 122–134. IEEE Computer Society, 2004.

Van Deursen, Arie. De software-evolutieparadox. http://homepages.cwi.nl/~arie/intreerede/, 2005. Inaugural lecture Delft University of Technology.

Van Deursen, Arie, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

Van Deursen, Arie and Paul Klint. Little languages: Little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):pages 75–92, 1998.

Van Deursen, Arie, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):pages 26–36, 2000.

Van Deursen, Arie and Joost Visser. Source model analysis using the JJ-Traveler visitor combinator framework. *Software: Practice and Experience*, 34(14):pages 1345–1379, 2004.

Van Dijk, Hylke W., Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, volume 3047 of *Lecture Notes on Computer Science*, pages 203–221. Springer-Verlag, 2005.

Van Genuchten, Michiel. The impact of software growth on the electronics industry. *IEEE Computer*, 40(1):pages 106–108, 2007.

Van Ommering, Rob, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):pages 78–85, 2000.

Viennot, Gérard Xavier. Heaps of pieces, I: Basic definitions and combinatorial lemmas. In *Proceedings of the Colloque de combinatoire énumérative (UQAM 1985), Montreal, Canada*, volume 1234 of *Lecture Notes in Mathematics*, pages 321–350. Springer-Verlag, 1986.

Visser, Eelco. Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 216–238. Springer-Verlag, 2004.

Wang, Yingxu, Graham King, Hakan Wickberg, and Alec Dorling. What the software industry says about the practices modelled in current software process models? In *Proceedings of the 25th EUROMICRO Conference*, volume 2, pages 162–168. IEEE Computer Society, 1999.

Whittle, Jon and Johann Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323. IEEE Computer Society, 2000.

Wirth, Niklaus. Program development by stepwise refinement. *Communications of the ACM*, 14(4):pages 221–227, 1971.

Yin, Robert K. *Case Study Research: Design and Methods*. Sage Publications, 2003.

Summary

Two well-known software engineering laws state that 1) software has to be changed constantly in response to new user requirements or a changed environment, that is, software evolves continuously and 2) software that is changed, becomes more complicated. The consequence of this trend of increasing complexity is that the maintainability of software systems decreases over time: it becomes more and more difficult to make changes. The complexity of a software system is for a large part determined by its structure, often referred to as architecture. This thesis focuses on the evolution of software architectures. This type of evolution, while common, comes with a considerable risk and cost. Our aim is to reduce this risk and cost.

Two obvious strategies to remedy the problem of reduced maintainability of software systems are: 1) apply techniques to manage the increasing complexity, or 2) apply techniques to reduce the complexity. Automation and abstraction are two basic software engineering techniques to support these strategies. In this thesis we investigated the applicability of technologies for a new approach of software development, based on automation and abstraction, to support the evolution of software architectures. This new approach is referred to as model-driven software development.

The main research question addressed in this work is: *How can evolution of software architecture be supported?* We clarified the scope of our work by defining three related subquestions that deal with integration of evolution support in industrial practice, the implication of the use of product line principles on the evolution support, and the automation of evolution support using model-driven software development techniques.

To get a better understanding of the use software engineering technologies for different types of tasks in industry, we started by conducting a survey. In this survey we asked software practitioners of eight software development organisations about the software engineering technologies they use. We also paid attention to the situation in which certain technologies are applied and potential problems. The trends we observed during this survey include: the use of product-line approaches, the informal use of modelling and the importance of the evolutionary aspect of software (i.e., software is seldom developed from scratch). Partly the results of this survey motivated the aforementioned research questions.

Then, by case studies at Océ and ASML, we investigated how the evolution of software architectures can be supported. In particular we considered four types of software engineering tasks related to software evolution

Evaluation A first step when performing changes to a software system, is the evaluation of whether these changes can be realised within the current architecture. Here, we mainly investigated how such an evaluation can be conducted in the context of a software product line.

Conformance checking When an architecture has to be changed it is useful to know to extent to which it is consistent with other development artefacts. We focused on how model-driven software development technologies can be applied to answer that question.

Migration We investigated how an actual migration can be partly automated by the use of model transformations.

Documentation A disadvantage of the application of domain-specific languages for model-driven software development is that the definition of a (graphical) notation requires considerable effort. We investigated how model transformations can be deployed to map such languages to UML notation.

We studied each of these tasks separately in a case study.

The informal use of modelling in industry makes it necessary to introduce a normalisation step to enable the integration of evolution support in industrial practice. This thesis includes several examples of how to implement this step, which is typically context specific. Additionally by the use of several standards in the area of model-driven software development we further improve the potential integration of our results in practice.

In several chapters we address the impact of the use of product-line principles for the development of software systems on the software evolution support we introduce. Although the increased scope of software product-lines makes such support more difficult to develop, at the same time the return on investment (e.g., for the use of a model-driven approach) is much improved.

The model-driven support for the evolution tasks that we present in this thesis follows a similar three-step pattern. A set of source models is first 'preprocessed' into a form suitable for model transformations. This preprocessing includes a normalisation step as well as a translation into a representation based on the same technology as the model transformations. Then, model transformations are applied that are defined in a model transformation language. These transformations do the actual work such as conformance checking (i.e., using two source models) or migration. Finally, the resulting target models are postprocessed into the desired target form. This might be a graphical representation or some representation intended for further processing.

Samenvatting

Twee bekende wetten in de software-engineering zeggen dat: 1) software voortdurend moet worden aangepast aan nieuwe en gewijzigde omgevings- en gebruikerseisen; met andere woorden software evolueert continu en 2) software die gewijzigd wordt, wordt steeds ingewikkelder. Het gevolg van deze toenemende complexiteit is dat de onderhoudbaarheid van softwaresystemen afneemt met de tijd: het wordt steeds moeilijker veranderingen aan te brengen. Voor een groot deel wordt de complexiteit van een softwaresysteem bepaald door zijn structuur, ook wel architectuur genoemd. De focus in dit proefschrift is op de evolutie van softwarearchitecturen. Hoewel dit soort evolutie vaak voorkomt, is ze risicovol en kostbaar. Ons doel is het risico en de kosten die gepaard gaan met de evolutie van softwarearchitecturen te verminderen.

Voor het probleem dat de onderhoudbaarheid van softwaresystemen afneemt bestaan twee voor de hand liggende oplossingsstrategieën: 1) het gebruik van technieken om de toenemende complexiteit te beheersen en 2) het gebruik van technieken om de complexiteit te verminderen. Automatisering en abstractie zijn twee bekende software-engineeringtechnieken die voor deze twee strategieën ingezet kunnen worden. In dit proefschrift hebben we onderzocht hoe technieken voor een nieuwe aanpak voor software ontwikkeling, die gebaseerd is op automatisering en abstractie, toegepast kunnen worden voor de evolutie van softwarearchitecturen. Deze nieuwe aanpak wordt modelgedreven softwareontwikkeling genoemd.

De hoofdonderzoeksvraag die we behandelen is: *Hoe kan de evolutie van softwarearchitecturen worden ondersteund?*. Een drietal gerelateerde subvragen bakenen ons onderzoek verder af. Deze subvragen gaan over de integratie van potentiële evolutieondersteuning in de industriële praktijk, de gevolgen van het gebruik van productlijnen op de evolutieondersteuning en de automatisering van de evolutieondersteuning door middel van modelgedreven softwareontwikkeltechnologieën.

Om beter te begrijpen welke softwareontwikkeltechnologieën op welke manier worden ingezet voor verschillende soorten taken, zijn we begonnen met een enquête. In deze enquête hebben we mensen in verschillende rollen bij een achttal organisaties in de software-industrie gevraagd naar de ontwikkeltechnologieën die ze gebruiken. Belangrijke aandachtspunten hierbij waren ook de situatie waarin gebruik wordt gemaakt van een bepaalde technologie en de problemen die daarbij optreden. Enkele trends die we tijdens de enquête hebben opgemerkt zijn: het toenemende gebruik van productlijnen, de informele wijze van modelleren en het belang van het evolutionaire aspect van software (softwaresystemen worden zelden vanuit het niets ontwikkeld). Deels hebben de resultaten van deze enquête geleid tot eerder genoemde onderzoeksvragen.

Vervolgens hebben we met behulp van casestudy's bij Océ en ASML onderzocht hoe de evolutie van softwarearchitectuur ondersteund kan worden. We hebben ons hierbij gericht op vier typen softwareontwikkeltaken die te maken hebben met software-evolutie

Evaluatie Een eerste stap bij wijzigingen is het vaststellen of zij binnen de huidige architectuur gerealiseerd kunnen worden. Hier hebben we vooral onderzocht hoe zo'n evaluatie in de context van een productlijn gemaakt kan worden.

Consistentie Wanneer de architectuur moet worden aangepast is het nuttig te weten in welke mate deze consistent is met andere ontwikkelartefacten. Wij hebben ons geconcentreerd op de mogelijkheden van modelgedreven softwareontwikkeltechnieken voor het vinden van een antwoord op die vraag.

Migratie Wij hebben onderzocht hoe een daadwerkelijke migratie gedeeltelijk geautomatiseerd kan worden met behulp van modeltransformaties.

Documentatie De toepassing van domeinspecifieke talen voor modelgedreven softwareontwikkeling heeft als nadeel dat de ontwikkeling van een (grafische) notatie veel inspanning vergt. Wij hebben onderzocht hoe model transformaties kunnen worden ingezet om zulke talen af te beelden op de UML notatie.

We hebben elk van deze taken bestudeerd in een aparte casestudy.

De informele wijze van modelleren in de industrie maakt het noodzakelijk een normalisatiestap te introduceren om de integratie van evolutieondersteuning in de industriële praktijk mogelijk te maken. Dit proefschrift bevat verscheidene voorbeelden van hoe deze stap, die contextafhankelijk is, te realiseren. Verder verbeteren we de integreerbaarheid van onze resultaten in de praktijk door het gebruik van een aantal standaarden op het gebied van modelgedreven softwareontwikkeling.

In meerdere hoofdstukken komen we terug op de invloed van het gebruik van productlijnen voor de ontwikkeling van softwaresystemen op de evolutieondersteuning die we ontwikkelen. Alhoewel de grotere reikwijdte van productlijnen dit moeilijker maakt, nemen de mogelijkheden de noodzakelijke investeringen (bv. voor het gebruik van een modelgedreven aanpak) terug te verdienen eveneens toe.

De modelgedreven ondersteuning voor de software-evolutietaken die we in dit proefschrift presenteren volgt een vergelijkbaar driestappenpatroon. Een verzameling bronmodellen wordt eerst zodanig geprepareerd dat de modellen een formaat krijgen dat geschikt is om te transformeren met modeltransformaties. De preparatiestap omvat een normalisatie en een vertaling naar een representatie die gebaseerd is op dezelfde technologie als de toe te passen modeltransformaties. Vervolgens worden deze transformaties, die gedefinieerd worden in een modeltransformatietaal, toegepast. Deze transformaties doen het eigenlijke werk, zoals het controleren van consistentie (dus met twee bronmodellen) of een migratie. Tenslotte wordt het resultaat nog bewerkt om het in een gewenst formaat te brengen. Dit kan bijvoorbeeld een grafische representatie zijn of een tijdelijke representatie bedoeld voor verdere bewerking.

Curriculum Vitae

Bas Graaf was born in The Hague on Friday the 13th of January in 1978. There, he graduated *gymnasium* (high school) in 1996 at CSG Overvoorde. Subsequently he enrolled in the computer science program at Delft University of Technology. After specialising in software engineering he received his master's degree in 2002. He wrote his master's thesis on component-based software development and the Unified Modeling Language under supervision of Dr. P.G. Kluit and Prof.dr.ir. J.L.G. Dietz.

In 2002 he started his Ph.D. research at the software technology department of Delft University of Technology. During this research under supervision of Prof.dr. A. van Deursen he investigated the applicability of model-driven software development technologies to support software evolution.

Titles in the IPA Dissertation Series since 2002

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural

Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of

Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development*